Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

An Optimization Theory for Structured Stencil-based Parallel Applications

Massimiliano Meneghin

Supervisor
Marco Vanneschi

10 January 2010

Abstract

In this thesis, we introduce a new optimization theory for stencil-based applications which is centered both on a modification of the well known owner-computes rule and on base but powerful properties oftoroidal spaces. The proposed optimization techniques provide notable results in different computational aspects: from the reduction of communication overhead to the reduction of computation time, through the minimization of memory requirement without performance loss.

All classical optimization theory is based on defining transformations that can produce optimized programs which are computationally equivalent to the original ones. According to Kennedy, two programs are equivalent if, from the same input data, they produce identical output data.

As other proposed modifications to the owner-computes rule, we exploit stencil application feature of being characterized by a set of consecutive steps. For such configurations, it is possible to define specific two phase optimizations.

The first phase is characterized by the application of program transformations which result in an efficient computation of an output that be easily converted into the original one. In other words the transformed program defined by the first phase is not computational equivalent with respect to the original one.

The second phase converts the output of the previous phase back into the original one exploiting optimized technique in order to introduce the lowest additional overhead. The phase guarantees the computational equivalence of the approach.

Obviously, in order to define an interesting new optimization technique, we have to prove that the overall performance of the two phases sequence is greater than the one of the original program.

Exploiting a structured approach and studying this optimization theory on stencils featuring specific patterns of functional dependencies, we discover a set of novel transformations which result in significant optimizations.

Among the new transformations, the most notable one, which aims to reduce the number of communications necessary to implement a stencil-based application, turns out to be the best optimization technique amongst those cited in the literature.

All the improvements provided by transformations presented in this thesis have been both formally proved and experimentally tested on an heterogeneous set of architectures including clusters and different types of multi-cores.

To Augusta

The art of programming is the art
of organizing complexity,
of mastering multitude and
avoiding its bastard chaos as
effectively as possible

Notes on Structured Programming,
Edsger Dijkstra

Acknowledgments

First of all I would like to thank my supervisor Marco Vanneschi. A couple of years has already passed since I met him the first time at the beginning of my PhD in Pisa. In this time I had the privilege of working with him, and equally appreciating his outstanding approach to research and his incomparable way of sharing his structured knowledge with people around him.

I would like to thank Prof. Giovanni Gaiffi, an extremely brilliant mathematician who got the patience of introducing myself into the extraordinary world of toroidal spaces. The enthusiasm I felt working with him gave me an important support for getting to the end.

I am grateful to Prof Marco Danelutto, Prof. Marco Aldinucci, Dr. Massimo Torquati, Dr. Carlo Bertolli, Prof. Massimo Coppola, and Dr. Andrea Lottarini for all the interesting discussions we had on stencils and HPC issues. They always found time to listen to me and to provide good advices.

I would like also to thank all people that shared happy times during those years; from the guys of "*mensa piccola*" to the guys of the HPC laboratory, the crazy guys of "*via Svezia*" and obviously the evergreen group of "*Vittorio2*". We had a lot of fun together. Between all of them, I am particularly grateful to Valeria, Doriana e Stefano who are like a family for me.

I can not forget one of the person I am most attached to. Thank you Francesca for the great times we spent together and for being always at my side when I need you; I am glad to have a fantastic sister like you.

Finally I would like to lovely thank my parents that always support and encourage me to face serenely and courageously all the challenges of life. Most of what I reached comes from your teaching.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

DATA PARALLELISM is a well known paradigm of parallel programming, which is characterized by replication of functions and partitioning of data over a set of virtual processing nodes.

One of the most powerful data parallel paradigms is represented by the stencil concept. The class of stencil-based applications arises in many scientific fields. Stencils are exploited in explicit time-integration methods for the numerical solution of partial differential equations (for example in climate, weather, ocean modelling [47, 2, 42, 49]) or in finite difference time domain methods for computational electromagnetic [51] and for multimedia and image processing applications [54, 21].

A stencil-based data parallel application is characterized by a certain number of iteration steps; at each step the processing nodes calculate a new value for all the elements of the partitioned domain, complying with some functional dependencies over sets of elements. Two of the main issues in stencil-based applications are communications and caching. In parallel architectures, especially targeting fine grain parallelization, in order to reduce the computation time, the communications required to resolve data dependencies between different domain partitions can represent an unavoidable lower bound [46, 16, 44]. The other important issue is about extracting features from the stencil computational kernel that can be exploited to take full advantage of memory hierarchies, especially those of new multi-core architectures [48, 45, 20, 23, 15, 29].

Generally speaking, a model is a tool exploited to manage a representation of the object of study. The action of modelling is equivalent to defining a set of bounds in order to outline the peculiar characteristics of the object. The bounds are exploited to search model properties that can be useful in the study. The more the bounds are strict the more the features of the object are highlighted, but the less general the model is.

In the classic theory of program optimization, the objects of study are program transformations which can produce equivalent programs featuring different characteristics from the original one; for example, higher performance, lower memory requirements, or hopefully both.

The bound that is mainly exploited by the classic optimization model is the computational equivalence concept. This definition is a critical aspect because it is the yardstick by which we evaluate the legality or the safeness of a program transformation. Computational equivalence as given by Allen and Kennedy [24] states that two computations are equivalent if, from the same input, they produce identical values for the output variables at the time the output statements are executed, and the output statements are executed in the same order.

All the theory of dependency analysis, which is at the core of a huge and most important class of optimizations for parallelism and cache management, is based on a fundamental property of the classic optimization model. This property, which represents a sufficient condition for equivalence, asserts that a transformation is classified as legal when it changes only the execution order of the code while preserving every dependency.

In parallel environments, one of the most famous technique which is exploited to implement data parallel applications is the owner-computes rule. This defines a strategy in order to map the computation on the processes that implement an application. The owner-compute rule can be easily modeled as the natural extension of the data dependency optimization theory as it preserves all the data dependencies of a program.

1.1 Contributions and Motivation of Thesis

In this thesis we introduce a new optimization theory which is centered on a redefinition and an extension of the owner-computes rule. The proposed optimization techniques provide notable results in different computational aspects: from the reduction of communication overhead to the reduction of computation time, through the minimization of memory requirements without performance loss.

We develop an optimization theory *targeting exclusively stencil-based applications featuring specific patterns of functional dependency.* This further bound, which is not present in the classic approach, allows us to make the computational equivalence constraint less strict.

The specific stencil-based applications that we are targeting are characterized by a set of computational steps, $\{s_1, \ldots, s_n\}$, which can be modelled as unitary pieces of computation. At each step, new values for the data structure elements are computed complying eventually with some defined functional dependencies.

By exploiting techniques of the classical optimization theory, we can derive an equivalent data parallel program which is characterized by a sequence of optimized steps: $\{s_1^{opt}, \ldots, s_n^{opt}\}$. Because of the principle of preserving data dependencies, which is exploited by both data dependency theory and the owner-computes rule, it comes out that each optimized step s_i^{opt} is computational equivalent to the original one s_i. In other words, both s_i^{opt} and s_i return the same identical result from the same input state.

Our work focuses on the definition of a family of new program transformations which takes advantage from the two simple observations on stencil-based applications.

I A stencil-based application is usually composed by a set of steps.

II The computational equivalence between each steps (s_i) and its optimized version (s_i^{opt}) is a sufficient but not necessary condition for the safety of a program transformation.

We can therefore split the problem of defining new data parallel optimization techniques into two distinct phases.

I A study of transformations that can be applied to each step in order to obtain an output that with some additional operation can be transformed into the original one.

II An analysis of same mechanisms that can reduce the overhead of the operations required to reconstruct the original output. As we prove in thesis, it is possible to make this overhead negligible and in same cases to completely overturn it.

Concerning the first phase, we are interested in transformations that result in optimized program whose output differs from the original one only for the spatial position of the values in the data structures. In order to model this kind of difference, we define the concept of *'relaxed computational equivalence'*. A transformation is classified as relaxed-legal, or also relaxed-safe, when it is legal in the classic assertion, except for a spatial rearrangement of the values in the output data structures.

To give a clue about the spatial rearrangement to which we have alluded, we consider the Jacobi stencil in one dimensional space; a well known method for partial differential equations. The stencil, represented in figure 2.21(a), works by updating at step i the values of all the domain elements with the mean of their left and right element values at the previous step $i-1$. We call the left and right elements the stencil *shape* (colored in gray) while the central element (with a dashed fill style) is the *application point*. In fig. 2.21(a) J_{in} represents the working domain values at step i while J_{out} contains those at step $i+1$.

Along with the "original Jacobi", we consider also a "modified Jacobi", which is represented in fig. 2.21(b); it has the same function as the original stencil and also the same shape, except for the location of the application point which is shifted to the right by one position.

Now, let $L_{in} = \{2, 4, 8, 16, 24, 32, 64\}$ be a vector, representing a toroidal input domain, that we compute with one step of both the original and modified Jacobi. Figures 2.21(c) and 2.21(d) report the resulting vectors and also highlight the different stencil data dependencies. It is evident that the two output vectors are not equal; in other words the two results are different. Incidentally, they are strongly

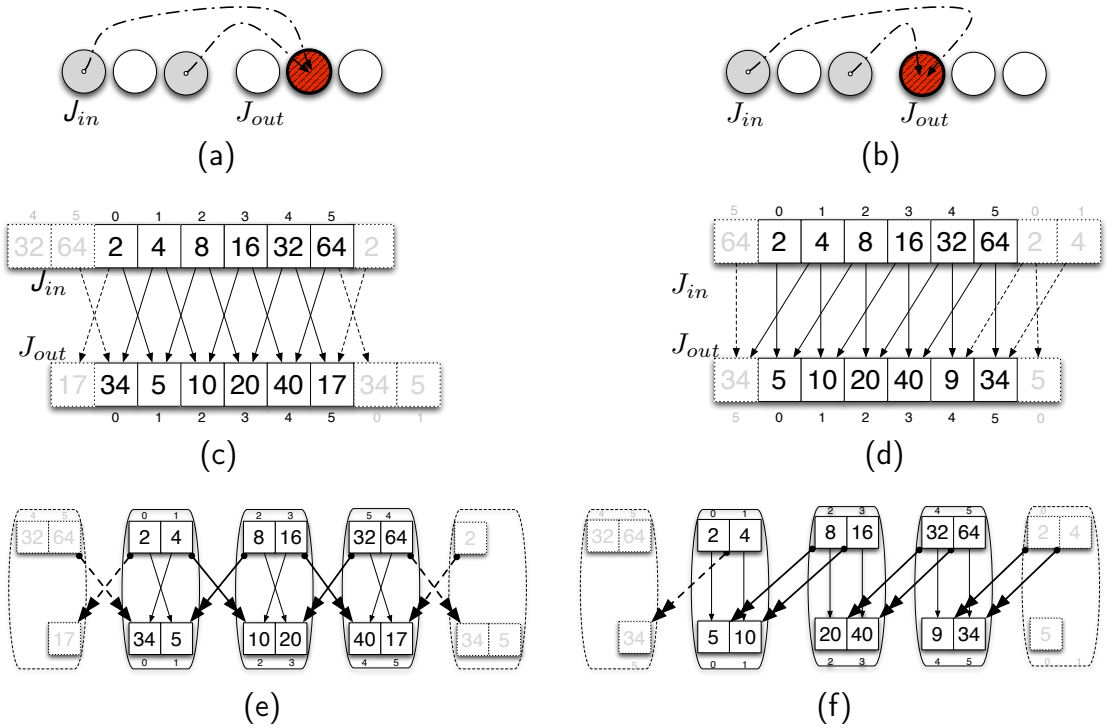


Figure 1.1: Representations, in one dimensional space, of both the Jacobi stencil (fig. 2.21(a)) and its modified version (fig. 2.21(b)), where the application point has been shifted by one position. Example of one step computation of the original (fig. 2.21(a)) and modified Jacobi (fig. 2.21(b)) over a circular vector representing a toroidal domain. Finally a parallelization of the original (fig. 2.21(e)) and modified (fig. 2.21(f)) Jacobi to highlight the different communication patterns.

related to each other: the values are the same, but stored in different positions in the data structure. More formally, a linear transformation is sufficient to transform one into the other and vice-versa.

Neglecting the different results in a first approximation, we consider a parallel implementation of both the previous stencils, where the input and output vectors are scattered over four processing nodes as presented in figs. 2.21(e) and 2.21(f).

The key point is now to analyze how the two stencil functional dependencies impact differently on communications. One single node, in the case of the original Jacobi, features incoming and outgoing communications with both left and right neighbours. In the case of the modified Jacobi, because of the different application point, one resource receives data only from the right neighbour and sends data only to the left one.

In conclusion, although the flow of exchanged data is the same in both cases, i.e. the number of elements sent or received does not change, the modified Jacobi

halves the number of incoming and outgoing communications per step, which has a direct impact on computation time, especially for fine grain parallelizations.

Analyzing the example with the classic optimization theory, a transformation that returns the modified Jacobi from the original one is not legal; but according to our new theory, we can assert it is classified as relaxed-legal. In fact, as previously noted, a linear transformation is sufficient to describe how to reconstruct the output of the original Jacobi by moving the elements of the other left or right.

The simple transformation we have presented on the Jacobi example is an interesting food for thought in relation to the first phase of the two step optimization schema that we have previously introduced. Nevertheless, smart mechanisms to reconstruct the original output with the low overhead still have to be defined for the second phase along with a proof of the performance gain of the overall approach.

It is worth mentioning that, because of the definition of relaxed-safe transformations, the second step consists in a rearrangement of the values into their right position in the data structures. As we are going to prove, such operations can be efficiently implemented exploiting fundamental characteristics of the toroidal spaces.

By adopting a structured approach and studying this two step optimization strategy on stencils featuring specific patterns of functional dependency, we discover a set of novel transformations which result in significant optimizations for a wide class of real world stencils:

- $\mathcal{Q}-transformations$ provide optimizations on the communication overhead, exploiting the relaxed computational equivalence and breaking the limitations of the well known owner-compute rule. With the transformations presented, the number of communications required to implement a generic stencil, defined over an n-dimensional space, can be reduced to n, with respect to $2 * n$ which is the best result achieved by solutions cited in the literature [46]. We also study mechanisms to reduce the overhead of taking the output working domain back to its original space ordering. In some cases we can even provide a zero reordering overhead, while still preserving the reduction of communication overhead.

  Experimental tests on clusters and multi-core architectures prove that $\mathcal{Q}-transformations$ offer more performance than other implementations. In a worst case where communication with all neighbours are required, the reduction of the communication overhead can be quantified relative to a "naive" implementation as a gain in time of up to 4.5; we define the time gain parameter as the ratio between the completion time of a reference implementation, i.e. the "naive" one, and the studied one, i.e. the implementation optimized with $\mathcal{Q}-transformations$.

- $\mathcal{QSF}-transformations$ are a formalization and a notable extension of these techniques which exploit replication of data to reduce communications [16]. One side effect of studying $\mathcal{QSF}-transformations$ is the definition of politics

for memory hierarchy management in the stencil computational kernel. The technique, which is based on a revisit of loop fusion classic optimizations, provides a time gain of up to 2.1, in a sequential environment.

- $\mathcal{QM} - transformations$ are optimizations targeting the reduction of memory constraints. This reduction is obtained without performance loss; rather, the type of memory access defined by $\mathcal{QM} - transformations$ provide notable performance benefit. With experimental results, we prove that $\mathcal{QM} - transformations$ almost halve the memory constraints and in some cases provide a time gain of up to 2.2.

1.2 Classical Optimization Theory

As claimed by Aho et al. [3], in order to target parallelism, mechanisms are required which help to reason about the dependencies among different dynamic executions of the same statement, in order to determine if they can be executed on different processors simultaneously.

In the classical theory of optimisation, the field of data dependence analysis provides the previous tools. By exploiting techniques based on data dependency, it is possible to transform an input program into a computationally equivalent program which features higher performance. Earlier studies of data dependence for vectorization purposes were by Lamport [35, 36] in parallel with Kuck, Muraoka and Chen [31, 43].

Other optimizations, along with a technique based on data dependence analysis, exist such as *ghost expansion* and *shift* methods. Nevertheless they all share the same underlying concept: **the transformation of the program always results in a computationally equivalent one**.

It is the aim of this Section to present firstly the concept of computational equivalence and then to introduce some of the most known transformations derived by classical optimization theory.

1.2.1 The Concept of Computational Equivalence

A program transformation has the goal of defining a new program which, while featuring the same "meaning" as the original one, provides different characteristics; obviously we are especially interested in better performance. In other words the transformed program has to do the same things as the original one but in a different way.

This raises the question of what behaviours the transformed program has to preserve. If the aim is to preserve the same run time, the transformed program cannot feature different performance from the original one, hence we would not be interested in the program transformation any more. What has to be preserved are the effects of the program.

Bacon et al. [8] present well and argue the logical path towards the definition of a criterion of equivalence between programs. They start from the following simple definition:

> A transformation is **legal** if the original and transformed programs produce **exactly the same output** for identical executions.

With some examples, the authors keep digging into the problem and introduce the issue of the relation between program transformations and correctness. The problem is to manage cases where the original program is not semantically correct.

The requirement for the transformed program to produce the same results would be an unnecessary bound when the original program is semantically incorrect. The authors therefore introduce the semantic correctness of the original program as an unavoidable condition in the definition of computational equivalence.

Finally Bacon et al. analyze the issue of floating point operations. Because of the finite precision of floating point representations, the request for bitwise identical results can be too strict. Therefore, they consider for the equivalence definition a certain degree of tolerance with respect to floating point operations.

We have reported this discussion about the concept of computational equivalence because it represents the point of departure from the classical theory of the new theory that we present in this thesis.

The classic approach is based on a model where a transformation is safe if the output data structures of a semantically correct program and the transformed one are **exactly the same**, apart from a tolerance for floating point values. Considering the output data structures of the original and transformed programs, according to the classical approach both data structures store the same value in a given position: the output values feature the same spatial organization in the output data structures of the two programs.

Our approach diverges from the foundation of the classic approach. Indeed we target a relaxed type of computational equivalence: we require that the data structures of the two programs store the same values but not necessarily with the same spatial organization.

1.2.2 The Mechanisms of Data Dependency

One of the most important strands in the classical optimization theory is the data dependence analysis. Data dependence is a relation between two statements which occurs if and only if

I both statement access the same memory location and at least one of them stores data in it, and

II there is a feasible run-time execution path from one statement to the other.

All data dependence analysis is based on the *fundamental Theorem of Dependence* of which we report the terms.

> Any reordering transformation that preserves every dependence in the program preserves the meaning of the program

A reordering transformation is defined as a program transformation which merely changes the execution order of the code, without adding or deleting executions of any statements.

The data dependence analysis mainly studies how to define transformations which, while reordering program execution and keeping the data dependency preserved, can produce programs featuring better performance than the original ones. Obviously, to prove the correctness of these transformations, a set of mechanisms to model data dependencies are necessary.

The mechanism used to represent dependence information on a piece of code is a dependence graph, where each node corresponds to a statement and arches indicate the existence of a dependence between two statements.

Usually the dependence graph is exploited to model linear code which does not feature iteration constructs. In the analysis of loops, which is a more complicated and interesting problem, other specific mechanisms are used.

When facing loop iterations, each statement inside a loop can be executed many times and therefore it is necessary to describe dependencies that can rise between different loop iterations. These kinds of dependencies are called *loop-carried dependencies*.

In order to model these more complex situations, Kuck [32] and Wolf [57] studied two mechanisms: respectively *distance* and *direction vectors*. The first models the dependency distances for the entire iteration, where the distance represents, in terms of iteration index, the computations between the execution of two statements that are linked by a dependency.

In some cases it is not possible to define at compile time the exact dependency distance or it is possible that the distance is not a constant. In these situations, the direction vectors are commonly used to partially characterize the dependencies.

The mechanisms presented are exploited to define and prove most of all known transformations of classical optimization theory based on data dependencies. In the following, we will review the most important ones.

Before proceeding, we wish to point out an issue with dependency analysis: the problem of automatically detecting when two array references may refer to the same element in different iterations. In analyzing a loop nest, usually a compiler tries some tests, which rely on the fact that expressions are almost linear. When dependencies are found, the compiler models them with distance or direction vectors. There are a large variety of tests, all of them aiming to prove independence in some cases. Indeed the direct problem of finding dependencies is a NP-complete problem [9].

1.2.3 Most Important Classic Transformations Based on Data Dependency

Loop Skewing

Loop skewing [35, 57] is a transformation that reshapes the iteration space to highlight existing parallelisms. The transformation changes the visit pattern in such a way that more operations can be computed in parallel. Skewing was invented to handle wavefront computations, where array updates propagate like a wave across iteration space.

Loop Tiling or Blocking

Stencil computation features global iterations through a data structure that is typically much larger than the capacity of the available data cache. Reorganization to take full advantage of the memory hierarchies has been studied. These investigations have mainly focused on *tiling* techniques [1, 17, 34, 55] which attempt to exploit locality by performing operations on cache-sized blocks of data before moving to the next block. Tiling optimizations are also known as *blocking* techniques.

Loop Fusion

A well known transformation aiming to modify the computation grain inside a loop is *loop fusion* [56], also known as *jamming*. The main concept of the transformation is to replace multiple loops with a single loop containing all statements. This optimization reduces the overhead of loop management and can increase both parallelism and data locality.

1.2.4 The Owner-Computes Rule and its Extensions

The owner-computes rule is a well known strategy to map computation on a set of processes, especially exploited in non shared memory model, which was first developed at Rice University for Fortan D, an HPF compiler [12]. The rule became one of the most exploited compilation schema for most of the HPF compilers.

The owner-compute rule is based on the concept of data ownership. After the data distribution phase, each process gets the ownership of the distributed data that it is storing; this means that it is the only one that can modify those data. The owned elements are always up-to-date and need no communication or synchronization before being accessed by the process.

Usually the rule is expressed with respect to assignment statements that defines updates of variables in a program: the process that owns the left-hand side (LHS) element is in charge of performing the calculation. Therefore a processor is the only one that can update its owned data and moreover is responsible for performing all

the operations in order to define the new values. In case those operations required non owned data, communications between processes are necessary.

Following the owner compute rule, a compiler can define for each one of the processors implementing a parallel application the computations and also the communications or synchronizations it has to perform.

An extension of the previous mapping schema is represented by the owner-stores rule [38]: the processor that is in charge of performing the computation for the new value of a variable is one of those owning a right-hand side (RHS) element. Because typically the RHS elements are distributed between more processors, there no just one way to apply the owner-stores rule. Because the owner-stores rule is still based on the ownership mechanism, a communication between the element that perform the new value computation and the owner of the variable is necessary when the two processes do not coincide.

The two schema, depending on the data distribution, can lead to different patterns of communication and usually the one that minimize the communication overhead is selected.

The optimization techniques that we are going to present can be modelled as an optimized case of the owner-stores rule for a specific kind of data parallel computations. Indeed, the operation of storing the value back to the right left-hand side location is performance one at the end of the computation, within what we call second phase, rather than at each step.

If we analyze the problem of computation mapping in the comprehensive picture for developing mechanisms to implement optimized data parallel applications, we can not miss the important and strict relation between the computation mapping with data mapping, i.e the data distribution. If there is a mismatch between the two components, i.e. data needed for computation is not assigned to the same process as the computation, communications are necessary.

The problem of both automatic data and computation mapping for scientific application has been discussed extensively in literature and as source of reference we report a work of Kennedy and Kremer [25], where the author provide a well structured classification of the solution in literature with respect to the dynamic and static approaches. All the presented solutions are heuristics, indeed most of the aspect of automatic data mapping are be NP-complete problems as proved by the following works.

Li and Chen proved that the problem of determining an optimal static alignment between the dimension of distinct arrays is NP-complete [39]. Anderson and Lam showed that the dynamic data layout problem is NP-hard in presence of control flow between nests loop [6]. Mace proposed three different formulations of dynamic data layout for interleaved memory machines [41]. Kremer that the inter-phase data layout is an NP-complete problem [27]. Bouchiette et al. showed that aligning temporaries array expressions even without common sub-expressions is NP-complete [10].

At this point we think worthwhile to go back and analyze again the Jacobi

example, while keeping in mind the relation between the data mapping and the computation mapping. The transformed Jacobi has been presented as modification of the owner-computes rule, indeed it can be modelled, as we commented before, as a particular case of owner-store rule: optimized Jacobi application exploit e different computational mapping with respect to the naive implementation, which instead follows the owner-computes rule.

There something more interesting in the feature of this particular application and in general in the specific data parallel applications that we are targeting. The set of data structured that the application uses each step is simple: two identical data structures one for the step input values, therefore used only as read-only data, and one for the step output values, used only for write operations. The data mapping therefore consists in distributing two identical vectors between the processes.

In such specific configuration we can observe that the same benefit of the previous presented optimization, which exploit the owner-store rule, can be reached modifying the data mapping while leaving the computation one unchanged, i.e. exploiting the owner-computes rule. Indeed, we can distribute J_{out} with a different alignment with respect to J_{in}: the process that stores $J_{in}[i]$ stores $J_{out}[i+1]$. With a such distribution we get the same communication pattern of the other optimized Jacobi version.

All the transformation that we are going to present feature the previous characteristic. Nevertheless we are going to describe them as particular cases of the owner-store rule for two main reasons. First we can easily study the new transformations analyzing the properties of the stencil shape, while introducing the partitioning only in a second time. Secondly we can use use the same formalism to describe optimization also optimization for sequential programs, where there is no data mapping.

What distinguish our work from the previous ones on data and computation mapping are the different approaches and results. Indeed, we focus on a restrict specific set of data parallel computations, which nevertheless represents the core of a lot of important scientific applications. Secondly we exploit an approach typical of parallel structured programming method and finally the presented transformation lead to important optimization results which provide an important extension to the ones already known in literature.

1.2.5 Optimizations not Based on Data Dependency

We report some important optimizations for stencil-based applications. Such transformations do not come from dependency theory but are still based on the same concept of computational equivalence.

Message shifting

Plimpton [46] first presented a technique, which we refer to as the $shift$ method, to reduce communications with diagonal neighbours in stencil computations. The $shift$ method cuts down the total number of sent messages per step from $3^n - 1$, which is the maximum number of neighbours in an n-dimensional space, to $2 * n$.

The technique is based on indirect communications with diagonal neighbours. A single step can be seen to be composed of a set of micro-steps, one for each space dimension. Differently from a *naive* method, the $shift$ technique requires, in order to support micro-steps, the interleaving of send and receive operations within the same step. This characteristic can produce some difficulties while managing the overlapping of communications and computations.

Oversending

Ding and He [16] propose what we call the *oversending* method but is also known as the *ghost expansion* method. They target the reduction of communications per step exploiting a kind of oversending technique. The idea is to expand the data size exchanged between processes in such a way that communications are not required at each step.

From a high level point of view, the *oversending* method is an optimization technique which uses the data parallel paradigm, that is based on the distribution of data and replication of functions, along with the mechanism of data replication.

The technique forces a processing unit to compute its own partition and in some steps to update previously received data instead of communicating. With this method, the mean amount of data exchanged per step is equal to the *naive* and $shift$ ones, but the mean number of send and receive operations per step is reduced.

The $shift$ and *oversending* methods can also be interpreted as stencil transformations, where one step of the modified stencil is a kind of macro-step that corresponds to two or more steps of the original stencil.

A stencil transformed with the *oversending* method can also be optimized with the $shift$ technique in order to delete diagonal communications as presented in Ding and He's paper.

Moreover the authors propose an optimized *ghost expansion* method for PDE problems which, by manipulating some algorithmic aspects of a PDE solver, reduces the mean data size exchanged.

Palmer and Nieplocha [44] summarize both previous methods and present the result of their implementations on different distributed architectures exploiting the message passing library *MPI*. The main result is that no single algorithm provides optimal performance on all platforms.

1.3 Structured Parallel Programming

Structured parallel programming exerts a great influence on our work.

The research field of structured parallel programming aims for a definition of a high level parallel programming environment where programmers use mechanisms to describe their applications according to a set of parallel patterns. Therefore the parallel paradigm of computation is the object of study in this field.

Essentially, two main concepts can be used to define parallel computation patterns: functions, which model the flow of operations, and data structures, which model the values of operations.

We can define two distinct classes of patterns according to their relation with the two previous components.

I One class is based on patterns that are focused only on the characteristics of the functions of a computation. We call this the functional pattern class.

II Another class, which is the most interesting because of its complexity, is composed of patterns which model features of a computation that take into account both functions and data structures. This class groups all the data in the parallel paradigm and therefore we call it the data parallel pattern class.

The first class has been the subject of studies dealing with an approach based on skeletons [14], which can be considered to be the forerunner of the structured one. The most studied parallel constructs in the functional pattern class are *farm* and *divide&conquer*.

I The *farm* skeleton is based on the principle of functional replication. The same computation has to be independently performed over a set of independent tasks. Hence, the computation is replicated over a set of processes which receive, according to some distribution strategy, the tasks that have to be computed.

   The function representing the computational task is exploited as a black box. Thus the farm model does not incorporate any knowledge about the data structures used by the functions.

II The *divide&conquer* skeleton is based on the *divide et impera* paradigm. A problem, that has to be resolved, is recursively broken down into two or more sub-problems, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

   As in the farm case, the data structures used to represent the problem are not modelled directly by the skeleton which is still mainly based on functional replications.

Another construct that the skeleton approach brings into the analysis is the *map*. This pattern represents the easiest case amongst the elements of the data parallel pattern class.

III The *map* skeleton belongs to the data parallel paradigm, where both functional replication and data partitioning are exploited. The map represents the easiest case of data parallelism, where no functional dependencies are defined between elements.

This is the only case of the three skeletons where the data structures of applications become a first class object of the model.

In all cases, the skeleton approach models the parallel patterns as second order functions.

Despite farm, divide&conquer and map structures, there are a set of real world applications which are not modelled by these skeletons. An example are stencil-based applications which are represented by data parallel patterns featuring functional dependencies that have to be resolved though communications. Casanova et al. [13] give a well structured presentation and analysis of some data parallel algorithms, highlighting the importance of the topology of the reference parallel architecture model.

In the case of the data parallel paradigm, which features functional dependencies between elements, the approach of defining the paradigm as a second order function has major limitations. Indeed, a specific skeleton would be required for each specific pattern of data parallelism: a skeleton for the Jacobi stencil, one for the Laplace stencil and so forth. Although the previously cited stencils feature a common generic pattern, the skeleton approach fails in its representation. Hence, attacking the problem of modelling a data parallel application according to the skeletal approach, implies a loss of generality and expressiveness.

The structured approach was defined with the aim of merging functional patterns, data parallel patterns and their compositions. The approach does not fail in the way that the skeletal one does, because it represents parallel patterns not as second order functions, but in terms of basic mechanisms of computation and data management.

In the field of data parallelism, High Performance Fortran (HPF) [40], represents the most important work and most optimization theories based on data dependency analysis are linked to this project; for instance the well known "owner-computes rule" has been defined for the HPF compiler developed at Rice University [12].

The goal of HPF was the parallelization of sequential Fortran code, but HPF cannot be classified in the structured research field because it is completely centered on the data parallel paradigm and does not take into consideration functional concepts.

As a reference structured environment, we consider Assist [52], whose language provides a definition of applications according to functionals, data parallel forms

and their compositions. Assist, in its data parallel component, has been influenced by HPF, indeed the actual Assist model is strictly based on the owner-computes rule.

All compilers associated with previous parallel programming environments are based on the classical theory of optimization: in other words they refer to the classical concept of computational equivalence.

In our work, by studying a extension of the owner-computes rule based on two phase technique, we discover a set of optimizations that can also be exploited to extend the compilers of both Assist and HPF or other structured environments. In the study of these new optimizations, we borrow the approach of the structured parallel programming methodology: we focus on stencils featuring specific patterns of functional dependencies. The relaxation of computational equivalence, which has been exploited in the one of the two phase in order to model a specific use of the owner-store rule, together with the restriction to specific patterns of dependencies, are essential hypotheses of all our reasoning. Both these bounds form the basis of a computational model where we can define and prove the set of properties that lead to new stencil transformations.

1.4 Plan of Thesis

The thesis is organized as follows:

Chapter 2: The Structured Stencil Model. We first present a formal model to capture the important features of a significant class of stencils. The model is called the "*structured stencil model*", where the adjective structured highlights not only the main role that geometric structures play in the model but also represents the purpose of setting our approach in the research field of structured parallel programming.

Secondly, we present a set of three examples, extracted from well known algorithms in the literature, to show how to represent stencils in the structured model. The examples have been carefully selected to demonstrate a wide range of different stencil features.

Then, the Chapter introduces an innovative in-depth classification of stencils. In order to characterize each class and capture its important features, as will be exploited in the rest of the thesis, a set of properties are presented and formally proved.

Finally, a specification of the structured mode is presented for the $\mathcal{HUA}$ stencil class, an entity which is relevant to our studies.

Chapter 3: The $\mathcal{HUA}$ Complete Architecture. We analyze the complete architecture of a framework for programming $\mathcal{HUA}$ stencil-based applications. The software architecture that we refer to is composed of four levels called

functional dependency, *partition dependency*, *concurrent*, and *firmware level*. At the *functional dependency level* a stencil-based application is represented in terms of the structured model, where the functional dependency is highlighted. At the *partition dependency level*, the $\mathcal{HUA}$ stencil description is translated in terms of space partitions. At the *concurrent level*, a representation of the stencil in terms of communicating processes is extracted. Finally at the *firmware level* the program is compiled for the particular architecture.

We analyze a particular stencil application at each level of the architecture. In the analysis, we provide a description of the best solutions in the literature for optimizing the number of communications in the implementation of a $\mathcal{HUA}$ stencil.

Chapter 4: $\mathcal{Q}-transformations$ One of the techniques most used to describe stencil computations is the *owner-computes* rule. Although the rule makes the definition of stencil parallelization an easier task, we discover that alternative solutions can lead to important new optimizations in our relaxed theory which reduce the number of dependencies between domain partitions.

The new transformations, which are classified as relaxed-safe, reduce to n the maximum number of communications required to implement a $\mathcal{HUA}$ stencil, where n is the number of dimensions of the targeted space. The optimizations achieved by the new techniques represent the best result compared to solutions cited in the literature.

Chapter 4: Step Fusion Transformations The scope of this Chapter is the introduction of a new and powerful class of stencil transformations called $\mathcal{Q}-\mathcal{S}tep-\mathcal{F}usion$ ($\mathcal{QSF}$).

Data parallelism is based on data distribution and function replication. A well known technique, called *ghost cell expansion*, aims to reduce the communication overheads by exploiting data replication. The transformation, which is expressed in the literature as a transformation at the concurrent level, introduces an interesting trade-off between a reduction in communication overheads and an increase in computational load.

The $\mathcal{QSF}-transformations$ expand and restructure the main concept of the *ghost cell expansion* up to the definition of a new class of stencil transformations which are defined at the functional dependency level instead of the concurrent one. Our new in-depth point of view of the *ghost cell expansion* technique makes possible the exploitation of the results of both $\mathcal{Q}-transformations$, for communication overhead reduction, and of the new optimizations which are focused on lowering computational load.

In the Chapter, we demonstrate analytically the benefit of $\mathcal{QSF}$–$transformations$ and we validate the results with a complete set of experiments on different kinds of computational architecture.

Chapter 6: Space Overlapping Transformations. We focus on the memory requirements for the implementation at the concurrent level of a $\mathcal{HUA}$ stencil.

Implementations of the $\mathcal{HUA}$ stencil which approximately halve the memory requirement to represent the working domain are well known, but they suffer from the drawback of increased computational load. Indeed these techniques require a copy operation for each element in the working domain.

In this Chapter, we present and formally prove the existence of specific $\mathcal{Q}-transformations$ called $\mathcal{QM}-transformations$. The stencil resulting from these new transformations can be associated with an in-place implementation which halves the memory requirements without introducing other computational overheads.

Chapter 7: Conclusions. We present the conclusions of the thesis and describe future work.

Appendix 4: $\mathcal{M}$ammu$\mathcal{T}$. This Appendix gives an in-depth introduction to $\mathcal{M}ammu\mathcal{T}$, the communication library that we used to test all our transformations on the Cell architecture.

Structured parallel programming is a parallel software development methodology which aims to deliver programmability, portability and interoperability, along with scalability and performance. To achieve these goals, it is important to define both a suitable set of high level parallel constructs and a communication language. The mechanisms of this language have to provide both very high performance and low overhead, for efficient implementation of parallel construct run time, and a clear cost model that allows the parallel construct composition to be optimized.

We describe our experience with defining abstract and concrete communication protocols optimized for structured parallel programming on single chip multi-core architectures. We implement and test our mechanisms on the IBM Cell BE chip multi-core. We detail a comprehensive cost model of the communications, which is a requirement for supporting automatic optimization in a structured parallel framework, and we report the achieved performance. Our implementation reaches best possible bandwidth and latency on this architecture: measured performance numbers are extremely close to actual hardware limits.

Chapter 2

The Structured Stencil Model

Abstract

To start our study of stencil-based applications, we need a well defined formalism to describe them and to demonstrate their properties.

In this Chapter, starting from some informal descriptions, we first present a formal model to capture the important features of a significant class of stencils. The model is called "*structured stencil model*", where the adjective structured highlights not only the main role that geometric structures play in the model but also sets our approach in the research field of structured parallel programming.

Secondly, we present a set of three examples, extracted from well known algorithms in the literature, to show how to represent stencils that exploit the structured model. The examples have been carefully selected to demonstrate a wide range of different stencil features.

Then, the Chapter introduces an innovative in-depth classification of stencils. In order to characterize each class and capture its important features, which will be exploited in the rest of the thesis, a set of properties are presented and formally proved.

Finally, a specification of the structured mode is presented for the $\mathcal{HUA}$ stencil class, which is an entity of relevance to our studies.

Contents

2.1 A Formalization of Stencils in a Structured Vision

We begin this Section with an informal description of each of the aspects of a stencil computation in order to prepare readers for the formal definition which follows.

2.1.1 An Informal Analysis of a Generic Stencil

Informally, a stencil-based application, or at least one of those that we are targeting in our studies, can be ideally represented as a working domain whose values are changed step by step according to some computations.

An example in a two dimensional toroidal space is the well known Jacobi iterative algorithm, which is based on finite difference approximations for solving partial differential equations. A description in pseudo-code of an application based on the Jacobi method is reported in Figure 2.1.

```
double J_in[10][10], J_out[10][10];                                  1
load_working_domain_values(J_in);                                    2
for(i_step = 0; i_step < 4; i_step++){                               3
                                                                     4
    forall((x, y) ∈ J_in){                                           5
        J_out[x][y] = (J_in[x][y + 1] + J_in[x][y − 1]               6
                     + J_in[x + 1][y] + J_in[x − 1][y])/4;           7
    }                                                                8
    swap(J_in, J_out);                                               9
                                                                     10
}                                                                    11
return_working_domain(J_out);                                        12
```

Figure 2.1: Description in pseudo-code of the Jacobi stencil application on a two dimensional toroidal space. To keep the notation light, we suppose the indices are automatically re-mapped onto the toroidal space, i.e. $J_{out}[-1,-1]$ matrix access is transformed into $J_{out}[9,9]$

The example features four iterations, which we call *steps* (in the literature they are sometimes also referred to as sweeps), over two matrices, which represent what we call the *working domain*. During a generic iteration i, the matrix J_{out} stores the new computed domain element values, while J_{in} stores the older ones, i.e. those computed in the previous iteration.

In the Jacobi algorithm, we use the **forall** construct (line 5 of the pseudo-code in Figure 2.1) to highlight the fact that no particular visit strategy is required in order

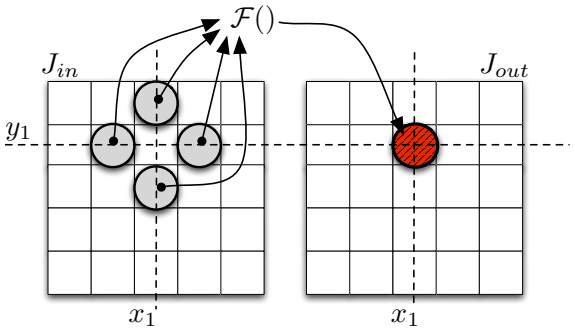


Figure 2.2: Representation of an application point and the corresponding shape which are featured by the Jacobi application whose description in pseudo-code is reported in Figure 2.1

to perform the update of the J_{out} matrix elements. In one single iteration, J_{in} and J_{out} can be accessed in whatever manner: row major, column major, or blocking. The result of the computation is correct in any case.

More formally, for a single iteration of the outermost cycle, i.e. the one controlled by the i_{step} variable, the statement inside the **forall** loop does not feature any data dependency with itself. Indeed, by definition [24], a necessary condition for a data dependency between two statements is that both of them access the same memory location and at least one stores data in it. Focusing on the Jacobi pseudo-code, the only write operations are computed on J_{out} memory locations where no read operation is performed. Therefore we can conclude that no data dependency is present and consequently no particular pattern of access is required.

Returning to the characteristics of the example of the Jacobi stencil-based application, we notice that it perfectly matches the informal description of a stencil computation that we gave at the beginning of this Section. Indeed, the Jacobi stencil consists in a working domain, which in turn is modelled by two matrices of real elements, whose values are changed according to the computation of the mean values over a set of real numbers. Two main entities thus have to be characterized in order to formally define a stencil: step and working domain.

The working domain hides a complex nature; we can highlight three important and distinct components: a *spatial structure*, a *computational domain* and an *evaluation map*.

I The spatial structure models the geometric features of the space where the elements of the stencil application are defined. In the Jacobi stencil, the spatial structure represents the space of valid indices that can be used to access the matrices J_{in} and J_{out}.

II The computational domain defines the possible values that can be associated to a generic element of the spatial structure; in the Jacobi example, the com-

putational domain is $\mathbb{R}$, i.e. the set of real numbers.

Before proceeding with the introduction of the last component of the working domain, we consider mandatory a comment about the generality of the formalism we are about to present. In our study, we confine ourselves to stencils featuring a particular structure, which is characterized by two main aspects.

1. We target those stencils that do not feature *loop-carried* dependencies inside a single step computation. Loop dependencies in the literature are classified as *loop-carried*, which arise because of the iterations of loops, and as *loop-independent*, which arise when two statements reference the same memory location within a single iteration of all their common loops.

   When no *loop-carried* dependency is detected, it means that there is no restriction on the possible visiting strategies that can be exploited to update all the elements of the working domain.

   Significant examples of the two different configurations are Jacobi and Gauss-Seidel algorithms. Both apparently have the same stencil shape, i.e. the functional dependencies of the two stencils are characterized by the same geometric representation, and also both of them feature a *loop-carried* dependency associated with the outermost loop, i.e. the one that is controlled by the i_{step} variable and models the iteration of the steps. Nevertheless, the Gauss-Seidel algorithm also presents a loop-carried dependency inside a single step. In order to clearly highlight the two behaviours, we report in Figure 2.3 a description in pseudo-code of an application based on the Gauss-Seidel method.

```
double J[101][101];                                              1
load_working_domain_values(J);                                   2
for(i_step = 0; i_step < 4; i_step++){                           3
    for(x = 1; x < 100; x++){                                    4
        for(y = 1; y < 100; y++){                                5
            J[x][y] = (J[x][y + 1] + J[x][y - 1]                 6
                + J[x + 1][y] + J[x - 1][y])/4;                  7
        }                                                        8
    }                                                            9
}                                                                10
return_working_domain(J);                                        11
```

Figure 2.3: Description in pseudo-code of the Gauss-Seidel stencil application in two dimensional space.

   The differences are evident when comparing the pseudo-code of the two stencil applications (see Figure 2.1 and Figure 2.3).

(a) The Jacobi example performs the computation exploiting two distinct matrices: one for the input values and one for the output values. In contrast, Gauss-Seidel exploits one single matrix as data structure; the input matrix is modified in place and then directly returned as the result.

(b) Another difference, which in some way is obviously linked to the previous point, consists in the fact that the Jacobi can be expressed as a *forall* cycle over all the elements of the working domain. Instead, in the case of Gauss-Seidel, we are forced to exploit *for* constructs to specify a particular visit of the matrix, i.e. in this case a row visit, in order to respect the loop-carried dependency associated with the single step.

This kind of behaviour is a characteristic of algorithms based on dynamic programming techniques, which also feature another important aspect. Usually these algorithms are provided with two main versions, one that features loop-carried dependency in the single step computation and the other that does not. This is the case for iterative methods for partial differential equations, where we face on one hand Jacobi and Red-Black methods, i.e. implementations that do not feature loop-carried dependencies, and on the other side Gauss-Seidel.

2. Another characteristic of the stencils which we are going to target concerns the spatial structure of the working domain; we require this entity to be an invariant with respect to the step computation. In other words, the stencil application is described by exactly one input and one output equivalent spatial structures, i.e. both data structures feature the same index space. We call this class of stencil *space invariant*.

   The Jacobi example falls into this category, but other stencil computations do not. For instance, a parallel *reduce* operation is characterized by steps featuring an input index space that is wider than the one that is associated with the output. Another example is the matrix multiplication algorithm, where we can define two, possibly different, input index spaces, i.e. those associated with the input matrices, and an output one, i.e. that associated with the resulting matrix.

   All the definitions of working domain presented and used in our study refer to the space invariant class. For completeness of our exposition, in Section 2.4, we introduce an informal discussion about an extension of the model in order to express also those stencils that do not belong to the *space invariant* class.

We now return back to the presentation of the last component of a working domain.

III The third component of a working domain is a map, in its mathematical definition, which defines a correspondence between the other two working domain

components; we call it the *evaluation map*. The map associates to each element of the spatial structure, i.e. to each pair of indices of the Jacobi matrix, one single value of the computational domain, i.e. an element of the set $\mathbb{R}$ of real numbers.

It could seem that the evaluation map is more or less equivalent to the mechanisms used in sequential programming languages to retrieve the value of an element in a multidimensional array. Those mechanisms are usually represented by the square bracket operator. For example, if we consider the element $e = (x, y)$ in a two-dimensional array J_{in}, the operation $J_{in}[e]$ returns at run time a value of the computational domain.

The evaluation map has a more sophisticated structure and plays a fundamental rule in our model; it is going to be a key component in most of the presented proofs of stencil properties. Three characteristic aspects distinguish the evaluation map from square bracket operator semantics.

I In an evaluation request through the map of our model, an instant time is a mandatory parameter. The selected temporal instant specifies, during the whole computation time, when the evaluation has to be computed. In other words, the temporal instant establishes the context for the evaluation.

II Even when the time instant is fixed, the evaluation map still has a different nature from the square bracket operator. The map does not return directly a value of the computational domain as the square bracket operator does. As we will see, the result of an evaluation is a formula that is going to be expressed by a couple of entities: a function and a set of spatial elements, whose evaluation results are going to be used as input parameters of the function.

III Finally, the computational map, as can be evinced from the previous points, is static information; it is invariant with respect to both the values of the input working domain and application run time.

An informal description of the evaluation map is not easy to present briefly because of its complexity. With the previous description we just wanted to give a clue about both the structure and expressive power of such a mechanism. For a complete and formal definition of the map, we refer to Definition 2.1.6, which describes the working domain component, and Definition 2.1.7, which describes the step component. Indeed, the two previous definitions result in a complete and formal description of the evaluation map.

The time dimension of the evaluation map forces the introduction of a time model in a stencil-based application. We consider the steps as those entities which beat out the time of a stencil-based computation. An interval in time between two consecutive steps establishes the context for the working domain evaluation.

By convention, we consider instant i the time before the beginning of the computation of step i and after the end of the computations of step $i - 1$. In this

scenario, steps are modelled as atomic operations. Indeed, the evaluation of an element of a spatial structure is defined only before or after a step and not during its computation.

Because in our study we focus only on space invariant stencils, we can assert the following property: for a generic step, the only possible difference between the input and output working domains is represented by the evaluation map. Indeed from the features of the space invariant stencil class, we know that the spatial structure does not change; it is an invariant component of our model. With this perspective, a stencil step can be described as the procedure which specifies how the evaluation map of the input working domain is changed into that of the output working domain.

2.1.2 Formalization of the Structured Stencil Model

Based on the previous informal description of stencil components, we proceed again along the same logical path to their formal definition.

Definition 2.1.1 (**Space Invariant Stencil**). A space invariant stencil ψ is defined by the following components:

$$\psi = \left(\mathcal{W}_\psi, \mathcal{T}_\psi\right) \tag{2.1}$$

$\mathcal{W}_\psi$ is called the **working domain** and $\mathcal{T}_\psi$ the **step set**.

A stencil therefore is simply modelled by two components, one representing the data of the computation and one concerning the specification of the computation. For instance, the Jacobi application (we associate the stencil to the abbreviation JCB) can be modelled as follows:

$$JCB = \left(\mathcal{W}_{JCB}, \mathcal{T}_{JCB}\right)$$

In the stencil structured model, the formal definition of the space invariant stencil component does not give useful information; it is just a container that identifies other components of a stencil.

To proceed with the formal definition, we have to analyze the two stencil components. We focus first on the step set.

Definition 2.1.2 (**Step Set**). Given a space invariant stencil $\psi = (\mathcal{W}_\psi, \mathcal{T}_\psi)$, its step set $\mathcal{T}_\psi$ is defined as an **ordered** set of steps:

$$\mathcal{T}_\psi = \left\{step_1^\psi, \ldots, step_g^\psi\right\} \tag{2.2}$$

Moreover, the set $\mathcal{C} = \{1, 2, \ldots, |\mathcal{T}_\psi|, |\mathcal{T}_\psi| + 1\}$, which is built from the indices of the ordered set $\mathcal{T}_\psi$, is called the **evaluation time set** and represents the temporal instants when an element of the working domain can be evaluated. Evaluating a working domain at time $\tau \in \mathcal{C}$ means evaluating it between $step_{\tau-1}^\psi$ and $step_\tau^\psi$ (see Figure 2.4). By definition the evaluation at time $\tau = |\mathcal{T}_\psi| + 1$ is the evaluation at the end of the last step and it represents the result of the stencil computation.

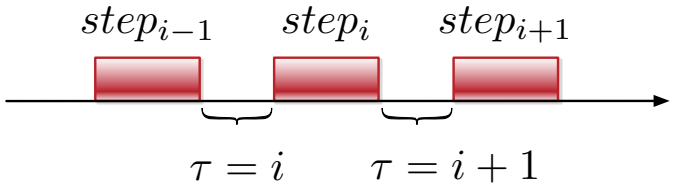


Figure 2.4: Representation of time instants in the structured stencil model. Evaluation of an element of a spatial structure are legal only between steps, which are considered as atomic operations.

The definition of the step set is almost trivial and leaves all the formal problems to the definition of the step component. Incidentally, an important aspect derives from the previous definition: the evaluating time set. It establishes the time instants of the computation, during which we can proceed to evaluate a working domain element. According to the definition, an evaluation can be performed only between two contiguous steps. The evaluation map, which is formally presented later within the step definition, is therefore going to be parametric with respect to a time component, modelled by an element of the evolution time set.

The application based on the Jacobi stencil, presented in Figure 2.1, is represented by a step set of four elements:

$$\mathcal{T}_{JCB} = \left\{step_1^{JCB}, step_2^{JCB}, step_3^{JCB}, step_4^{JCB}\right\}$$

In contrast to the formal definition of a stencil, the step set is not just a container of components. From its structure some important information can be derived to classify stencils. In the Jacobi case, $\mathcal{T}_{JCB}$ features the characteristic of being composed of a sequence of identical steps. Indeed, from the application pseudo-code, we claim that the computations associated with each step are equal: $\forall i, j\ step_i^{JCB} = step_j^{JCB}$. In other words, if the order of the elements inside the step set is changed, the result of the computation remains the same. We analyze in depth and formalize the semantics of the equality relationship between steps in Section 2.3.1.

Concerning the evaluation time set, in our example we have $\mathcal{C}_{JCB} = [1, 5]$. Therefore, according to the rules given by the model, it is possible to retrieve a parametric evaluation of an element of the working domain only before beginning the computation ($\tau = 1$), before the beginning of a step ($\tau = 1, 3, 4$), or at the end of the computation ($\tau = 5$).

Before metaphorically digging into the definition of the step components, we need to formalize the working domain which was used in the step set definition. As it is going to be clear at the end of the whole model formalization, the different stencil components are strictly linked to one another. Therefore it is not possible to give independent definitions of each of them; it happens that the definition of one component uses another component and vice versa. This was the case for the step set and working domain, where we had to give the definition of the first while exploiting the second, without this last having previously been defined in a formal

way. Introducing the spatial domain before would have led to the same situation.

To introduce the working domain, we start with the formalization of one of its three components, the spatial structure.

Definition 2.1.3 (**General Spatial Structure).** Let ψ be a space invariant stencil over an n-dimensional space. We define the **spatial structure** $\mathcal{M}_\psi$ associated with ψ as a subset of $\mathbb{N}^n$.

The definition collects a wide range of possible spatial structures; it includes both sparse and dynamic ones. While most of the results of our studies can be extended to general spatial structures, in the rest of the document we confine our work to regular ones.

Definition 2.1.4 (**Regular Spatial Structure).** Given a space invariant stencil ψ over an n-dimensional space, we define a **regular spatial structure** $\mathcal{M}_\psi$ a particular compact and convex subset of $\mathbb{N}^n$ containing the origin. $\mathcal{M}_\psi$ is the result of n cartesian products of natural intervals:

$$\mathcal{M}_\psi = \overbrace{[0, m_1 - 1] \times [0, m_2 - 1] \times \ldots \times [0, m_n - 1]}^{n\ factors}$$

The vector $m = (m_1, \ldots, m_n)$ is named **length vector**. A generic element e of $\mathcal{M}_\psi$ is a vector of components $(e_1, \ldots, e_n)$ defined with respect to $(\epsilon_1, \ldots, \epsilon_n)$, which is the standard base of $\mathbb{N}^n$.

Although the formal definition of the regular spatial structure can appear complicated, its meaning is straightforward. The definition selects from all the general structures those that are represented by the index space of a multi-dimensional matrix. The space of the indices is represented by the Cartesian products of the index intervals along each single dimension. Each component of the length vector defines the number of items that the index space has along the associated space dimension.

Using again the Jacobi-based application (see Figure 2.1) as a tutorial example, the working domain is given by two 10×10 matrices, therefore its spatial structure is represented as $\mathcal{M}_{JCB} = [0, 9] \times [0, 9]$. For each element $e = (x, y) \in \mathcal{M}_{JCB}$ we have $0 \leqslant x \leqslant 9$ and $0 \leqslant y \leqslant 9$. Finally the JCB length vector is $m^{JCB} = (m_x, m_y) = (10, 10)$.

Actually, the spatial model that we have just given does not exactly match the spatial structure of the Jacobi example, indeed it does not capture the toroidal feature of the space. Because we are going to extract some important properties from toroidal spaces, we present also the definition of a regular spatial structure mapped onto these particular spaces.

Definition 2.1.5 (**Toroidal and Regular Spatial Structure).** Given a space invariant stencil ψ over an n-dimensional toroidal space, we define a **toroidal regular spatial structure** $\mathcal{M}_\psi$ as the result of n cartesian products as follows:

$$\mathcal{M}_\psi = \overbrace{\frac{\mathbb{Z}}{m_1} \times \frac{\mathbb{Z}}{m_2} \times \ldots \frac{\mathbb{Z}}{m_n}}^{n\ factors}$$

The symbol $\frac{\mathbb{Z}}{m_i}$ represents a one-dimensional toroidal space based on an m_i module. The vector $m_\psi = (m_1, \ldots, m_n)$ is named **module vector**.

The representation of regular toroidal spatial structures is equivalent to the regular structure; the characteristics of the index space are the same except for the toroidal property. Indeed, the space identified by the toroidal segment $\frac{\mathbb{Z}}{m_i}$ is equal to that represented by $[0, m-1]$, plus the additional toroidal feature.

The number of spatial points associated with the two spatial segments are equal, nevertheless the two associated index spaces are different. In the toroidal case, an infinite number of indices is associated with each spatial element, while in the other case there is a one-to-one correspondence between elements and indices. This new defined spatial structure is the correct model for the Jacobi working domain.

In our study, we treat the features of toroidal spaces and consequent optimizations as extreme cases for two main reasons.

I Obviously, the mechanisms for managing toroidal space can be used to represent all those applications which are naively defined on such distinguishing spaces. This aspect has limited impact because it is hardly difficult to face this kind of application in real world cases.

II Foremost, a wide range of applications can be remapped onto toroidal spaces, as we will show later. Thanks to the remapping mechanisms, certain non toroidal applications can exploit, in addition to their standard features, new properties that come from the new kind of target space.

We can now finally give a complete and formal definition of one of the two components defined by a stencil-based application:

Definition 2.1.6 (Working Domain). Consider a space invariant stencil ψ with regular spatial structure $\mathcal{M}_\psi$ and evaluation time set $\mathcal{C} = \{1, \ldots, g\}$. Then let $\mathcal{D}$ be the domain of the values associated with each element of the spatial structure; we name it **computational domain**. We define the **evaluation map** component as follows:

$$\begin{gathered} eval \colon \mathcal{M}_\psi \times \mathcal{C} \mapsto \mathcal{D}_\psi \\ eval(e, i) = d \end{gathered} \tag{2.3}$$

For a fixed stencil computation time instant $i \in \mathcal{C}$, the map $eval(e, i)$ associates to each element e of the spatial structure a value of the computational domain $\mathcal{D}$. Graphically, we define $eval(e, i) = \mathcal{M}^i[e]$.

Finally having defined the previous elements, a **working domain** $\mathcal{W}_\psi$ of a space invariant stencil ψ is given by the following components:

$$\mathcal{W}_\psi = \left(\mathcal{M}_\psi, \mathcal{D}_\psi, \mathcal{M}^i_\psi[.]\right) \tag{2.4}$$

Looking at the Jacobi application of the tutorial, we can therefore model the working domain component, according to the previous definition, as:

$$\mathcal{W}_{JCB} = \left(\mathcal{M}_{JCB} = \frac{\mathbb{Z}}{10} \times \frac{\mathbb{Z}}{10}, \mathbb{R}, \mathcal{M}^i_{JCB}[.]\right)$$

The spatial structure is toroidal and the computational domain is the set of real numbers. For the evaluation map, from the definition of the working domain, we can just describe its type, which is of the form:

$$eval^{JCB}\colon \left(\frac{\mathbb{Z}}{10} \times \frac{\mathbb{Z}}{10} \times [1,5]\right) \mapsto \mathbb{R}$$

In order to complete the whole formalization, the step component remains to be presented. Recalling that for space invariant stencil-based applications the only change that a step component can produce on the input working domain is the redefinition of a new evaluation map, we can introduce the following:

Definition 2.1.7 (**Space Invariant Step**)**.** Let $step^\psi_i$ be the i-th step of a space invariant stencil ψ, which is associated with a working domain $\mathcal{W}_\psi = (\mathcal{M}_\psi, \mathcal{D}_\psi, \mathcal{M}^i_\psi[.])$. $step^\psi_i$ defines parametrically the evaluation map for the spatial structure at application time instant $i+1$ as follows:

$$\begin{aligned}
\forall e \in \mathcal{M}_\psi \quad &\overset{step_i}{\rightarrow} \quad \left(\mathcal{F}_{(i,e)}, \mathcal{S}^\psi_{(i,e)}\right) \\
\mathcal{S}^\psi_{(i,e)} \quad &= \quad \left\{g_1, g_2, \ldots, g_k |\ \forall \alpha\ g_\alpha \in \mathcal{M}_\psi\right\} \\
\mathcal{F}_{(i,e)} \quad &: \quad \mathcal{D}^{|\mathcal{S}^\psi_{(i,e)}|} \mapsto \mathcal{D} \\
\mathcal{M}^{i+1}_\psi[e] \quad &= \quad \mathcal{F}_{(i,e)}(\mathcal{M}^i_\psi[g_1], \ldots, \mathcal{M}^i_\psi[g_k])
\end{aligned}$$

For the generic element e we call $\mathcal{S}_{(i,e)}$ the **stencil shape** and e itself the **application point** (**AP**). The set $\mathcal{S}^\psi_{\mathcal{M}^i} = \left\{\forall \mathcal{S}^\psi_{(i,e)} | e \in \mathcal{M}_\psi\right\}$ is named the **shape set** of $step_i$.

In a stencil representation with a sequential language, the application point coincides with the left hand side element of the inner loop statement, where the computation to update the domain element is expressed. Indeed, the application point specifies the position where the result of the computation is going to be stored.

There is therefore a strong correspondence between the application point and the well known "owner-computes" rule which can be summarized as follows:

Definition 2.1.8 (**Owner-Computes Rule).** In the parallelization of a computation, the processing element that owns the left-hand side element of the statement will perform the calculation.

Therefore, in our model, the previous definition can be revisited and rewritten as:

Definition 2.1.9 (**Owner-Computes Rule in the Structured Model).** In the structured model for stencil-based applications, the owner-computes rule establishes that during a parallel computation the processing node holding an application point is in charge of computing the new value.

The owner-computes rule is one of the most used techniques to implement data parallel applications, but, as will be shown in the rest of the thesis, sometimes it can be a limiting strategy which precludes some interesting optimizations for the reduction of both communication and computation overheads.

Coming back once more to the Jacobi tutorial case, we know that all the steps of the step set are equal, therefore it is sufficient to define one generic step of the set as presented in Table 2.1.

2D Jacobi STEP(JCB)

$$
\begin{aligned}
\forall e = (x,y) \in \mathcal{M}_{JCB} \quad & \overset{step_i^{JCB}}{\rightarrow} \quad \big(\mathcal{F}_{ie}, \mathcal{S}_{ie}^{JCB}\big) \\
\mathcal{S}_{(x,y)}^{JCB} \quad & = \quad \Big\{(x,y+1),(x,y-1),(x+1,y),(x-1,y)\Big\} \\
\mathcal{F}_{(i,e)} \quad & : \quad \mathbb{R}^4 \mapsto \mathbb{R} \\
\mathcal{F}_{(i,e)} \quad & : \quad \big(r_1,r_2,r_3,r_4\big) \mapsto \frac{1}{4}\big(r_1+r_2+r_3+r_4\big)\ \forall r_i \in \mathbb{R} \\
\mathcal{M}_{JCB}^{i+1}[e] \quad & = \quad \mathcal{F}_{(i,e)}\big(\mathcal{M}_{JCB}^{i}[(x,y+1)], \mathcal{M}_{JCB}^{i}[(x,y-1)], \\
& \quad\ \ \mathcal{M}_{JCB}^{i}[(x+1,y)], \mathcal{M}_{JCB}^{i}`[(x-1,y)]\big) \\
& = \quad \frac{1}{4}\big(\mathcal{M}_{JCB}^{i}[(x,y+1)] + \mathcal{M}_{JCB}^{i}[(x,y-1)] \\
& \quad\ \ +\mathcal{M}_{JCB}^{i}[(x+1,y)] + \mathcal{M}_{JCB}^{i}[(x-1,y)]\big)
\end{aligned}
$$

Table 2.1: Step component in the structured stencil model of the Jacobi stencil application described in pseudo-code in Figure 2.1

In the Jacobi step model, each element of the spatial structure is associated with the parametric description of a shape, which in turn defines, for the generic application point, the well known geometric cross form. It is worth mentioning that, although the Jacobi step model has been provided for the generic $step_i^{JCB}$ in a parametric way with respect to index step i, all the shape components, i.e. $\mathcal{S}_{(x,y)}^{JCB}$, are independent of parameter i. Therefore all the steps of the application feature the same shape set, which is, as we recall, the stencil component that groups all the shapes defined in the same step set.

Let us now turn our attention to the Jacobi step function. The component is defined from a four-dimensional input domain to a one-dimensional output one, where four is the cardinality of all the shapes of one step. The step function returns the arithmetic mean of the four components of an element belonging to the $\mathbb{R}^4$ domain.

Finally we consider the definition of the Jacobi evaluation map for time instant $i+1$. It is declared to exploit both the information of shape and of stencil function. From the example of the Jacobi step model, it becomes clear what the parametric definition of the evaluation map means to us . The evaluation of an element $e = (x, y) \in \mathcal{M}$ at time instant $i+1$ depends on the function $\mathcal{F}_{(i,e)}$ and on the evaluation at the previous time instant of those elements that are indicated by the shape $\mathcal{S}_{(x,y)}^{JCB}$. Although the exact value of an evaluation can be known only at run time, the evaluation map gives us the tools to prove system invariant properties.

The Jacobi structured model features a simple structure, but on the other hand it shows important characteristics of some main classes of stencils that we are going to present in Section 2.3. We briefly summarize the features as follows.

I The step set is composed of all equal elements, as demonstrated by the reported single definition of a step model and also by the fact that if we change the sequence of steps, we get the same result.

II The elements of all shapes can be identified through a linear translation (or equivalently affine transformation) of the corresponding application point. For instance, if we consider the element $\overline{e} = (x, y+1)$, we know that it belongs to the shape $\mathcal{S}_{(x,y)}^{JCB}$ and that it can be written as:

$$\overline{e} = (x, y+1) = \begin{bmatrix} x \\ y+1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

where (x, y) is obviously the element application point.

III The previously cited linear transformations are not completely general. The elements of a shape are described by a linear transformation where the matrix, usually called *rotation matrix*, is the identity.

The characteristic that each shape is equivalent to another except for a rigid translation means that each element in the working space geometrically features some functional dependencies.

In the formalization of the model, there is no restriction on the form of the $\mathcal{S}^{\psi}_{(i,e)}$ elements. The set can collect, for example, all the elements of the spatial structure or it can be the empty set. The $\mathcal{S}^{\psi}_{(i,e)}$ elements can depend on a specific application point or not; this means that, in the same step, we can have different shapes relative to different application points. From the previous consideration, we can claim that the formalization, for the class of space invariant stencils, is completely generic and does not present any limitation of expressiveness.

2.1.3 Considerations on the "Structured" Feature

Before proceeding with some examples which show how to exploit the previous formalism, we would like to focus the discussion a little on the *"structured"* adjective with which we have characterized our model. The are many different reasons to label the model with such an adjective.

I The "structured" adjective captures the structural nature of the logical path that we followed when defining, component after component, the whole model form. Indeed, we started defining a stencil as a couple of simple containers, i.e. the working domain and the step set, and then we recursively gave the definition of all the components that characterize a stencil.

II One of the components of the model is the spatial structure, which is exploited in the definition and formalization of most of the stencil components. This fact leads us to the conclusion that the spatial structure is undoubtedly at the core of our model. We consider the reference to the role of the spatial structure a justification for the use of the "structured" adjective.

III Foremost, the adjective highlights that our studies are collocated in the structured parallel programming research field.

The model at the core of structured parallel programing is based on a high level mechanism to express specific parallel patterns of a computation.

The strength of this approach is the complete hiding of the low level aspects of communication and synchronization from programmers. This comes from knowledge of the parallel pattern that can be exploited in the parallel implementation.

In our study we concentrate on well specified data parallel patterns. The restricted focus allow us to completely manage both the functions that are replicated and the data that are distributed. Data management especially is at the core of the definition of a new optimization theory based on a relaxed vision of computational equivalence: the fundamental concept for studying program transformations.

2.2 Tutorial Examples

In this section, we demonstrate the modelling of three stencil-based applications: Laplace (LPC), which is an edge detection algorithm from image processing, Red-Black (RB), which is a method for solving partial differential equations, and Floyd-Warshall (FW), which is a well known graph analysis algorithm for finding the shortest paths in a weighted, directed graph. The examples have been carefully selected to demonstrate a wide range of different stencil features.

2.2.1 Laplace

Edge detection is a problem of fundamental importance in image analysis. In typical images, edges characterize object boundaries and are therefore useful for segmentation, registration and identification of objects in a scene. A solution for the problem is the Laplacian operator: a high-pass filter which is mathematically represented by the result of the two-dimensional sum of the second derivatives of an image convolved with a Gaussian curve. The second derivatives detect rapid intensity changes and the Gaussian smooths out the effects of noise.

Consider now the example of an application which filters four times a 1024x1024 black and white image, exploiting the Laplace operator. For simplicity, we suppose that the image is defined over a toroidal space. A representation of the application in pseudo-code is reported in Figure 2.6.

The structure of the pseudo-code is similar to the Jacobi example, the main difference being the shape; the Laplace, as it is graphically represented in Figure 2.5, features one element more than the Jacobi shape.

Once again, we consider the hypothesis of the toroidal property of space where the application is defined. This hypothesis does not have any correspondence with real world cases of image processing. Indeed, input images hardly ever represent scenes that are toroidal. Nevertheless we model the Laplace by exploiting this

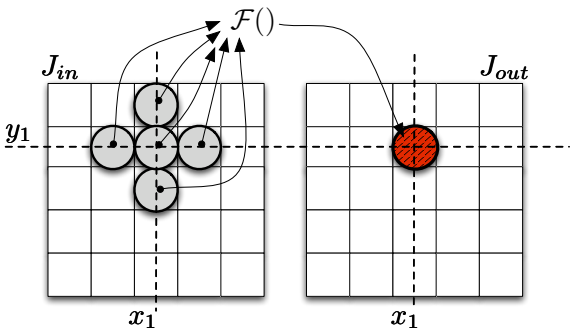


Figure 2.5: Representation of an application point and the corresponding shape which are featured in the Laplace application whose description in pseudo-code is reported in Figure 2.6

```
double J_in[1024][1024], J_out[1024][1024];                          1
load_working_domain_values(J_in);                                    2
for(i_step = 0; i_step < 4; i_step++){                               3
                                                                     4
  forall((x, y) ∈ J_in){                                             5
      J_out[x, y] = (J_in[x, y] + J_in[x, y + 1] + J_in[x, y − 1]    6
                    +J_in[x + 1, y] + J_in[x − 1, y])/4;             7
  }                                                                  8
                                                                     9
  swap(J_in, J_out);                                                 10
}                                                                    11
return_working_domain(J_out);                                        12
```

Figure 2.6: Representation in pseudo-code of a Laplace stencil application on a two-dimensional toroidal space. To keep the notation light, we assume the indices are automatically mapped onto the toroidal space, i.e. $J_{out}[-1, +1]$ is transformed to $J_{out}[+1023, +1]$

hypothesis for the sake of simplicity. We present and discuss in depth the problem of modelling applications based on non-toroidal spaces in Section 2.3.8, providing simple one-dimensional examples.

The structured model of the Laplace application is reported in Table 2.2. The observations that we can make are the same as those made for the Jacobi stencil example.

I The step set contains steps that are all equivalent. If their order changes, no differences can be noted.

II Each element of a shape is described by a linear transformation of the corresponding application point.

III All the shapes, defined in a step, feature the same geometric structure when analyzed using a coordinate system centred on the application point. This is a consequence of the fact that the rotation matrices of the linear transformations are all equal to the identity matrix.

2.2.2 Red-Black

We present an example which is based on a variant of the well known Gauss-Seidel relaxation method for the equation solver.

The Gauss-Seidel method is similar to the Jacobi method, except for a different relation with the working domain: Gauss-Seidel features data dependencies inside

2D Laplace (LPC) Model

$$
\begin{aligned}
LPC &= \left(\mathcal{W}_{LPC}, \mathcal{T}_{LPC}\right) \\
\mathcal{W}_{LPC} &= \left(\frac{\mathbb{Z}}{1024} \times \frac{\mathbb{Z}}{1024}, \mathbb{R}, \mathcal{M}^{i}_{LPC}[.]\right) \\
m_{LPC} &= (1024, 1024) \\
\mathcal{T}_{LPC} &= \left\{step_1^{LPC}, step_2^{LPC}, step_3^{LPC}, step_4^{LPC}\right\} \\
\mathcal{C}_{LPC} &= [1, 5]
\end{aligned}
$$

$$
\begin{aligned}
\forall e = (x, y) \in \mathcal{M}_{LPC} \overset{step_i^{LPC}}{\rightarrow} & \left(\mathcal{F}_{ie}, \mathcal{S}^{LPC}_{ie}\right) \\
\mathcal{S}^{LPC}_{(x,y)} &= \big\{(x, y), (x, y+1), (x, y-1), \\
&\quad (x+1, y), (x-1, y)\big\} \\
\mathcal{F}_{(i,e)} &: \mathbb{R}^5 \mapsto \mathbb{R} \\
\mathcal{M}^{i+1}_{LPC}[e] &= \frac{1}{4}\big(\mathcal{M}^{i}_{LPC}[(x, y)] + \mathcal{M}^{i}_{LPC}[(x, y+1)] \\
&\quad + \mathcal{M}^{i}_{LPC}[(x, y-1)] + \mathcal{M}^{i}_{LPC}[(x+1, y)] \\
&\quad + \mathcal{M}^{i}_{LPC}[(x-1, y)]\big)
\end{aligned}
$$

Table 2.2: Structured model of the Laplace (LPC) stencil application that is described in pseudo-code in Figure 2.6

```
double J_in[100][100], J_out[100][100];                                  1
load_working_domain_values(J_in);                                        2
for(i_step = 0; i_step < 4; i_step++){                                   3
                                                                         4
    forall((x, y) ∈ J_in){                                               5
        if((x, y) ∈ black)                                               6
            J_out[x, y] = (J_in[x, y + 1] + J_in[x, y − 1]               7
            +J_in[x + 1, y] + J_in[x − 1, y])/4;                         8
        else                                                             9
            J_out[x, y] = J_in[x, y]                                     10
    }                                                                    11
                                                                         12
    swap(J_in, J_out);                                                   13
    i_step++;                                                            14
                                                                         15
    forall((x, y) ∈ J_in){                                               16
        if((x, y) ∈ red)                                                 17
            J_out[x, y] = (J_in[x, y + 1] + J_in[x, y − 1]               18
                        +J_in[x + 1, y] + J_in[x − 1, y])/4;             19
        else                                                             20
            J_out[x, y] = J_in[x, y]                                     21
    }                                                                    22
                                                                         23
    swap(J_in, J_out);                                                   24
}                                                                        25
return_working_domain(J_out);                                            26
```

Figure 2.7: Representation in pseudo-code of a Red-Black stencil application on a two-dimensional toroidal space. To keep the notation light, we assume the indices are automatically re-mapped onto the toroidal space, i.e. for example $J_{out}[-1,-1]$ matrix access is transformed into $J_{out}[99,99]$.

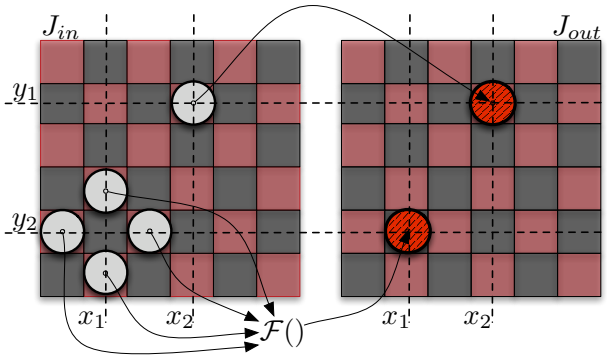


Figure 2.8: Representation of two application points and the corresponding shape which are featured by the Red step of the Black-Red application whose description in pseudo-code is reported in Figure 2.7.

Red Step of 2D Red-Black (RB) Model

$$\forall e = (x, y) \in \mathcal{M}_{RB} \quad \overset{black_i}{\rightarrow} \quad \left(\mathcal{F}_{ie}, \mathcal{S}^{RB}_{ie}\right)$$

$$\mathcal{S}^{RB}_{(x,y)} \quad = \quad \begin{cases} \big\{(x, y+1), (x, y-1), (x+1, y), \\ \quad (x-1, y)\big\} \textbf{ if } e \text{ is black} \\ \\ \big\{(x, y)\big\} \quad \text{if } e \text{ is red} \end{cases}$$

$$\mathcal{F}_{(i,e)} \quad : \quad \begin{cases} \mathbb{R}^4 \mapsto \mathbb{R} \text{ if } e \text{ is black} \\ \\ \mathbb{R}^1 \mapsto \mathbb{R} \quad \text{if } e \text{ is red} \end{cases}$$

$$\mathcal{M}^{i+1}_{RB}[e] \quad = \quad \begin{cases} \frac{1}{4}\big(\mathcal{M}^i_{RB}[(x, y+1)] + \mathcal{M}^i_{RB}[(x, y-1)] \\ +\mathcal{M}^i_{RB}[(x+1, y)] + \mathcal{M}^i_{RB}[(x-1, y)]\big) \text{ if } e \text{ is red} \\ \\ \mathcal{M}^i_{RB}[e] \end{cases}$$

Table 2.3: Structured Step model of the Red steps of the Red-Black (RB) stencil application described in Figure 2.7

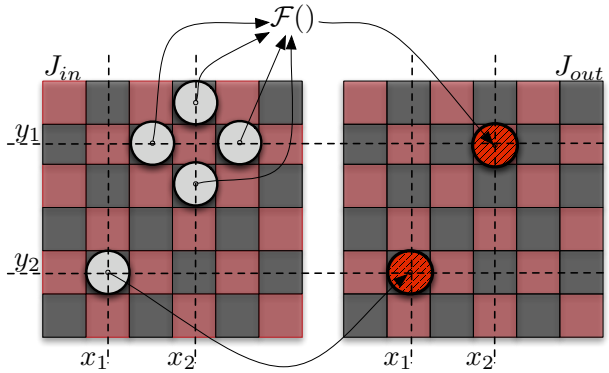


Figure 2.9: Representation of two application points and the corresponding shapes which are featured by the Black step of the Black-Red application whose description in pseudo-code is reported in Figure 2.7

Black Step of 2D Red-Black (RB) Model

$$\forall e = (x, y) \in \mathcal{M}_{RB} \quad \overset{red_i}{\rightarrow} \quad \left(\mathcal{F}_{ie}, \mathcal{S}^{RB}_{ie}\right)$$

$$\mathcal{S}^{RB}_{(x,y)} = \begin{cases} \{(x, y+1), (x, y-1), (x+1, y), \\ \quad (x-1, y)\} & \text{if } e \text{ is red} \\ \\ \{(x, y)\} & \text{if } e \text{ is black} \end{cases}$$

$$\mathcal{F}_{(i,e)} : \begin{cases} \mathbb{R}^4 \mapsto \mathbb{R} \text{ if } e \text{ is red} \\ \\ \mathbb{R}^1 \mapsto \mathbb{R} \text{ if } e \text{ is black} \end{cases}$$

$$\mathcal{M}^{i+1}_{RB}[e] = \begin{cases} \frac{1}{4}\big(\mathcal{M}^i_{RB}[(x, y+1)] + \mathcal{M}^i_{RB}[(x, y-1)] \\ +\mathcal{M}^i_{RB}[(x+1, y)] + \mathcal{M}^i_{RB}[(x-1, y)]\big) & \text{if } e \text{ is black} \\ \\ \mathcal{M}^i_{RB}[e] \text{ if } e \text{ is red} \end{cases}$$

Table 2.4: Structured Step model of the Black steps of the Red-Black (RB) stencil application described in Figure 2.7

the same step. The computations for updating element values during one step require values updated in the same step. In a single step, the computations are not independent, therefore it is mandatory to guarantee a specific matrix visit in such a way as to respects data dependencies. This kind of situation is usual in dynamic programming techniques to which Gauss-Seidel is strongly related.

An alternative to the Gauss-Seidel method is called Five-point Red-Black Gauss-Seidel relaxation and is based on the exploitation of a Red-Black ordering. The Red-Black is a particular ordering of matrix data structures, where elements are divided into red and black sets like in a chessboard, as represented in Figure 2.8 and Figure 2.9.

The Five-point Red-Black Gauss-Seidel relaxation method divides one step matrix update into two steps. In the first step, called the Black step, the method computes a Jacobi stencil only on black points, leaving unchanged the values of the red points. In the second step, called the Red step, the algorithm computes the Jacobi as well, but only on red points, this time leaving unchanged the black ones.

During a Black step, only red point values are read to update black elements (see Figure 2.8). Symmetrically, during the Red step, only black point values are exploited for computation of red ones (see Figure 2.9). Reassuming the structure of a generic phase, read operations are performed only on matrix elements of one color and write operations are performed on elements of the other color. Because no matrix element is the subject of both read and write operations in the same phase, we can claim that there is no data dependency.

As an example, Figure 2.7 reports a description in pseudo-code of an application that twice computes the Five-point Red-Black Gauss-Seidel relaxation method on a one hundred by one hundred matrix.

Before analyzing the pseudo-code and defining a structured model for the Red-Black application, we would like to focus the discussion on a well known characteristic of the algorithm.

Because of the Red-black ordering and its relationship to the Jacobi-like geometry of the stencil shapes, the five-point Red-Black Gauss-Seidel relaxation method features the characteristic that the computations of a generic step can be computed in place, i.e there is no need for two matrices, the input matrix is used also to store partial and final results.

In our analysis, we discard the possibility of capturing the *in-situ* computation feature for the following two reasons.

I We are interested in highlighting the formal nature of a stencil-based application through a structured model. For the sake of simplicity in the modelling phase, we purposely present a pseudo-code which, without considering the *in-situ* computation characteristics, explicitly exploits two matrices. Indeed, using this description, it is easier to observe the data relationships in order to define the evaluation map for the step model.

2D Red-Black (RB) Model

$$
\begin{aligned}
RB &= \left(\mathcal{W}_{RB}, \mathcal{T}_{RB}\right) \\
\mathcal{W}_{RB}j &= \left(\frac{\mathbb{Z}}{99} \times \frac{\mathbb{Z}}{99}, \mathbb{R}, \mathcal{M}^{i}_{RB}[.]\right) \\
m_{RB} &= (100, 100) \\
\mathcal{T}_{RB} &= \left\{black_1, red_2, black_3, red_4\right\} \\
\mathcal{C}_{RB} &= [1, 5]
\end{aligned}
$$

Table 2.5: Structure-based stencil model of the Red-Black stencil application described in pseudo-code in Figure 2.7. The step models of Red and Black steps are reported respectively in Table 2.3 and Table 2.4

II In addition, because the *in-situ* computation property can be statically detected, we claim that it is due to some compilation strategy that can be exploited for optimization purposes.

In the pseudo-code, the Black and Red phases are completely distinct, each one represented by a *forall* construct to underline the absence of data dependencies inside them. Inside the *forall*, an *if* construct is exploited to distinguish the computations that have to be associated to back or red elements. Because we do not exploit the *in-situ* computation feature, what exploits the *in-situ* feature in the algorithm is the operation of leaving some elements unchanged, i.e. black point elements during red steps and red point elements during black steps; in our pseudo-code this is implemented by a copy of the value from the input matrix to the output one.

The resulting application is a space invariant stencil that we can represent with the structured model as shown in Table 2.5, except for the step models that are presented in Table 2.3 for red steps and in Table 2.4 for black ones.

The structured model of the Red-Black application is more sophisticated than the Laplace and Jacobi ones. The first difference is the feature of the step set; it is composed of elements that are not all equal. The most evident difference is the step modelling, where the shapes and evaluation maps are decided according to the color of the application point.

```
double J_in[100][100], J_out[100][100];                                  1
load_working_domain_values(J_in);                                        2
                                                                         3
for(i_step = 0; i_step < 100; i_step++){                                 4
                                                                         5
    forall((x, y) ∈ J_in){                                               6
      J_out = min(J_in[x, y], J_in[x, i_step], J_in[i_step, y])          7
    }                                                                    8
                                                                         9
    swap(J_in, J_out);                                                   10
}                                                                        11
return_working_domain(J_out);                                            12
```

Figure 2.10: Description in pseudo-code of a Floyd-Warshall stencil application in a two-dimensional space.

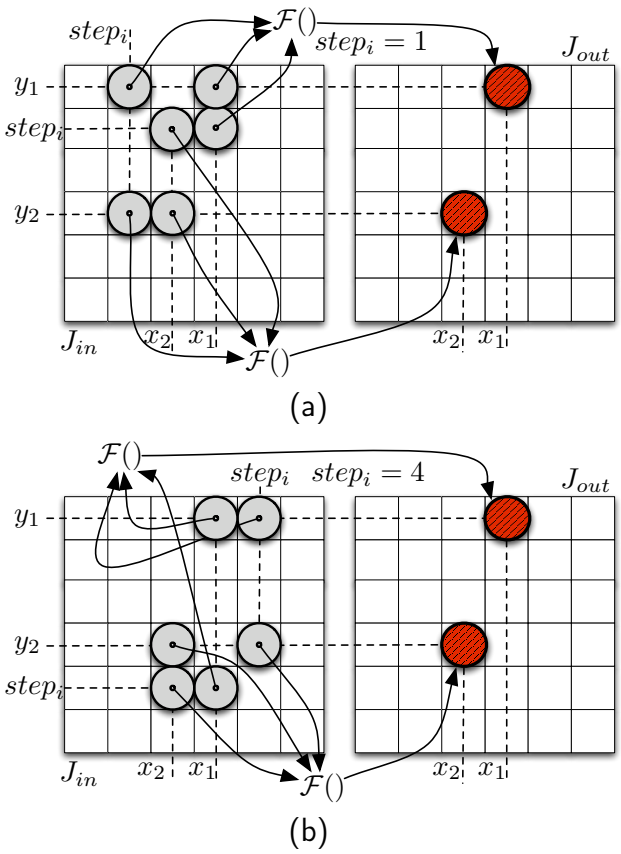


Figure 2.11: Representation of two application points and the corresponding shape which are featured, during iteration $step_1$ (Figure 2.11(a)) and $step_4$ (Figure 2.11(a)), by the Floyd-Warshall application whose description in pseudo-code is reported in Figure 2.10

2.2.3 Floyd-Warshall

The Floyd-Warshall algorithm is a graph analysis algorithm for finding the shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest paths between all pairs of vertices. The graph is usually represented by a matrix which stores the edge weights of an input graph. The method exploits a well known technique of dynamic programming and is based on the following recursive rule:

$$d_{i,j}^{k} = \begin{cases} w_{i,j} \text{ iff } k = 0 \\ d_{i,j}^{k} = \min\left(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}\right) \text{ iff } k > 0 \end{cases}$$

where w is the matrix of the graph edge weights. The result of the algorithm is stored in the matrix d^n, where n is the number of nodes in the graph.

We report a description in pseudo-code of the algorithm in Figure 2.10, where the Floyd-Warshall method is exploited to calculate the shortest paths of a graph featuring one hundred nodes. In the pseudo-code we continue to use the same notation used for the other examples instead of the classic Floyd-Warshall notation.

The application is a space invariant stencil that can be represented as reported in Table 2.6. The main feature of the structured model of the Floyd-Warshall application is the dependency of the shape element on the variable i, i.e. the step index. The variable defines the triangular access to the matrix typical of the Floyd-Warshall method.

2D Floyd-Warshall (FW) Model

$$
\begin{aligned}
FW &= \left(\mathcal{W}_{FW}, \mathcal{T}_{FW}\right) \\
\mathcal{W}_{FW}j &= \left([0,99]\times[0,99], \mathbb{N}, \mathcal{M}^i_{FW}[.]\right) \\
m_{FW} &= (100,100) \\
\mathcal{T}_{FW} &= \left\{step_1, \ldots, step_{100}\right\} \\
\mathcal{C}_{FW} &= [1,101]
\end{aligned}
$$

$$
\begin{aligned}
\forall e=(x,y)\in\mathcal{M}_{FW} &\overset{step_i}{\rightarrow} \left(\mathcal{F}_{ie}, \mathcal{S}^{FW}_{ie}\right) \\
\mathcal{S}^{FW}_{(x,y)} &= \left\{(x,y),(x,i),(i,y)\right\} \\
\mathcal{F}_{(i,e)} &: \mathbb{N}^3 \mapsto \mathbb{N} \\
\mathcal{M}^{i+1}_{FW}[e] &= \min\left(\mathcal{M}^i_{FW}[(x,y)], \mathcal{M}^i_{FW}[(i,y)] + \mathcal{M}^i_{FW}[(x,i)]\right)
\end{aligned}
$$

Table 2.6: Structured model of the Floyd-Warshall (FW) stencil application described in pseudo-code in Figure: 2.10

2.3 A Classification of Space Invariant Stencils

In this section, we present a classification of spatially invariant stencils which results from the analysis of the possible geometric or analytic characteristics of the various components of the structured stencil model. The same classification can be easily extended to non space invariant stencils, which are briefly presented in Section 2.4.

In Section 2.3.1 we report some definitions that compare operations on structured stencil components, which are then exploited in Section 2.3.2 and 2.3.2 to display a stencil classification based on the regularity properties.

2.3.1 The Equivalence Relation Between Stencil Components

In order to classify stencils, we need mechanisms to formally compare them. In this Section we present some definitions of equivalence between stencil components, such as step and shape.

The step is a complex stencil component: it defines the shape set, the association of a shape to each element of the spatial structure, and also the evaluation map at a certain time instant. We distinguish between two different types of equivalence, the first one is the following.

Definition 2.3.1 (**Structural Step Equivalence).** Two steps $step_i^{\psi}$, $step_j^{\psi}$ of the same space invariant stencil ψ are **structurally equivalent** when they feature the same set shape and both of them associate the same shape to each element of the spatial structure. We use the notation:

$$step_i^{\psi} \stackrel{\boxdot}{=} step_j^{\psi}$$

When two steps are structurally equivalent, all their elements feature the same functional dependencies between consecutive steps. Nevertheless, it is possible that, by exploiting two structurally equivalent stencils, the computation of an identical input working domain result in a different output working domain. In fact this kind of equivalence does not introduce any bounds on the computational functions associated with the evaluation map. In contrast, in the Jacobi example, all steps are structurally equivalent, but they also return the same results when they are applied to the same working domain. To guarantee that the two steps return the same result we have to introduce the following definition.

Definition 2.3.2 (**Complete Step Equivalence).** Two steps $step_i^{\psi}$, $step_j^{\psi}$ of the same space invariant stencil ψ are **completely equivalent** when they feature the same set shape and both of them associate the same shape and same function to each element of the spatial structure . We use the notation:

$$step_i^{\psi} \stackrel{\odot}{=} step_j^{\psi}$$

It is trivial to prove that two completely equivalent steps operating on the same input working domain produce identical results.

In the Red-Black example, all the Black steps are completely equivalent as well as the Red ones. In contrast, there is no structural or complete equivalence between the steps of the Floyd-Warshall method. Although we gave one single parametric description of all the application stencil steps, we used the index of the step set as a parameter in order to define the shapes. Indeed, analyzing a generic spatial structure element with respect to all the application steps, we find that the associated shape is always different. A graphical representation of this consideration is reported in Figure 2.11(a) and 2.11(b), where the shapes of two elements are pictured for two different steps.

Another component, whose equivalence relation will be important for the rest of the chapter, is the shape. With this type of equivalence we are interested in highlighting some geometric features which can link the elements of two different shapes:

Definition 2.3.3 (**Shape Equivalence).** Two shapes, which belong to the same shape set of a stencil ψ, are equivalent if, in a reference system centred on the associated application point, all the elements are defined by the same vectors. In other words $\mathcal{S}^{\psi}_{(i,e_1)}$ and $\mathcal{S}^{\psi}_{(i,e_2)}$ are equivalent if:

$$\mathcal{S}^{\psi}_{(i,e_1)} - e_1 = \mathcal{S}^{\psi}_{(i,e_2)} - e_2$$

The subtraction operation of component e from set $\mathcal{S}^{\psi}_{(i,e)}$ translates each element of the shape to a reference system centred on the application point. To highlight that two shapes are equivalent, we use the following notation:

$$\mathcal{S}^{\psi}_{(i,e_1)} \stackrel{\diamondsuit}{=} \mathcal{S}^{\psi}_{(i,e_2)}$$

A less formal view of the previous definition is that an observer, who analyzes the two shapes standing on their respective application points, does not notice any difference in the arrangement of the shape elements in the space.

In the Jacobi stencil application, we faced all equivalent shapes as well as in the Laplace application. In a generic Red-Black step, all the shapes associated with the same element color are equivalent, while in a single Floyd-Warshall step, there are no equivalent shapes.

2.3.2 A Classification Based on the Step Set Component

Next we move the focus of the discussion to the study of a stencil classification based on the geometric or analytic properties of the various components of the structured model. The first one that can be easily exploited for classifying stencils is the step set. In both Jacobi and Laplace method modelling, we highlighted that the step set is defined by a sequence of structurally equivalent steps. Therefore, a property for

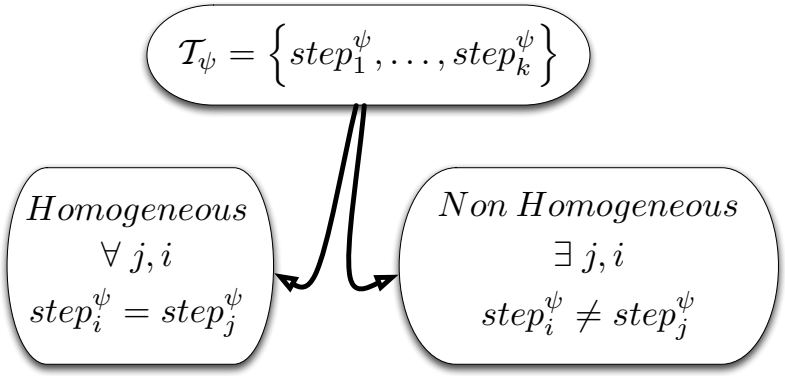


Figure 2.12: Stencil classification in the structured model based on the relations between the elements in the step set.

classifying stencils in different categories is the structural or complete equivalence of all the elements of the step set.

Definition 2.3.4 (**Homogeneous Stencil Class).** Let ψ be a spatially invariant stencil featuring $\mathcal{T}_\psi = \left\{step_1^\psi, \ldots, step_g^\psi\right\}$ as its step set. ψ is classified as an **homogeneous** stencil if all steps have one of the following properties:

I Structural homogeneity: $\forall step_i^\psi, step_j^\psi \in \mathcal{T}_\psi\ step_i^\psi \stackrel{\boxdot}{=} step_j^\psi$

II Complete homogeneity: $\forall step_i^\psi, step_j^\psi \in \mathcal{T}_\psi\ step_i^\psi \stackrel{\odot}{=} step_j^\psi$

Stencils that do not belong to this class are called **non homogeneous**.

A graphical representation of the stencil classes defined by the characteristics of the step set is reported in Figure 2.12.

Both Jacobi and Laplace stencils are classified as homogeneous stencils because they are composed only of structurally equivalent steps. The Red-Black stencil, instead, is non homogeneous because it features two different kinds of steps; ones for computations which update black points, i.e. Black steps, and ones for computations which update red points, i.e. Red steps. Likewise the Red-Black and Floyd-Warshall stencils are classified as non homogeneous; in the stencil step set there are no steps that are equivalent.

The property of complete equivalence is stronger than structural equivalence, indeed it is trivial to prove that the first property implies the second. From another prospective, we can say that the class of stencils that are structurally equivalent contains all the steps that are completely equivalent. In the rest of the thesis, we will discuss some optimizations of communication that require structural equivalence and some other optimizations of cache management that require the more strict complete equivalence. For simplicity, in the rest of the thesis, when we refer to equivalence between steps, without any other qualification, we will be referring to

structural equivalence. In this way, we can claim that all the properties that we will prove thanks to step equivalence are effective for both structural and complete equivalence.

2.3.3 A Classification Based on the Shape Set Component

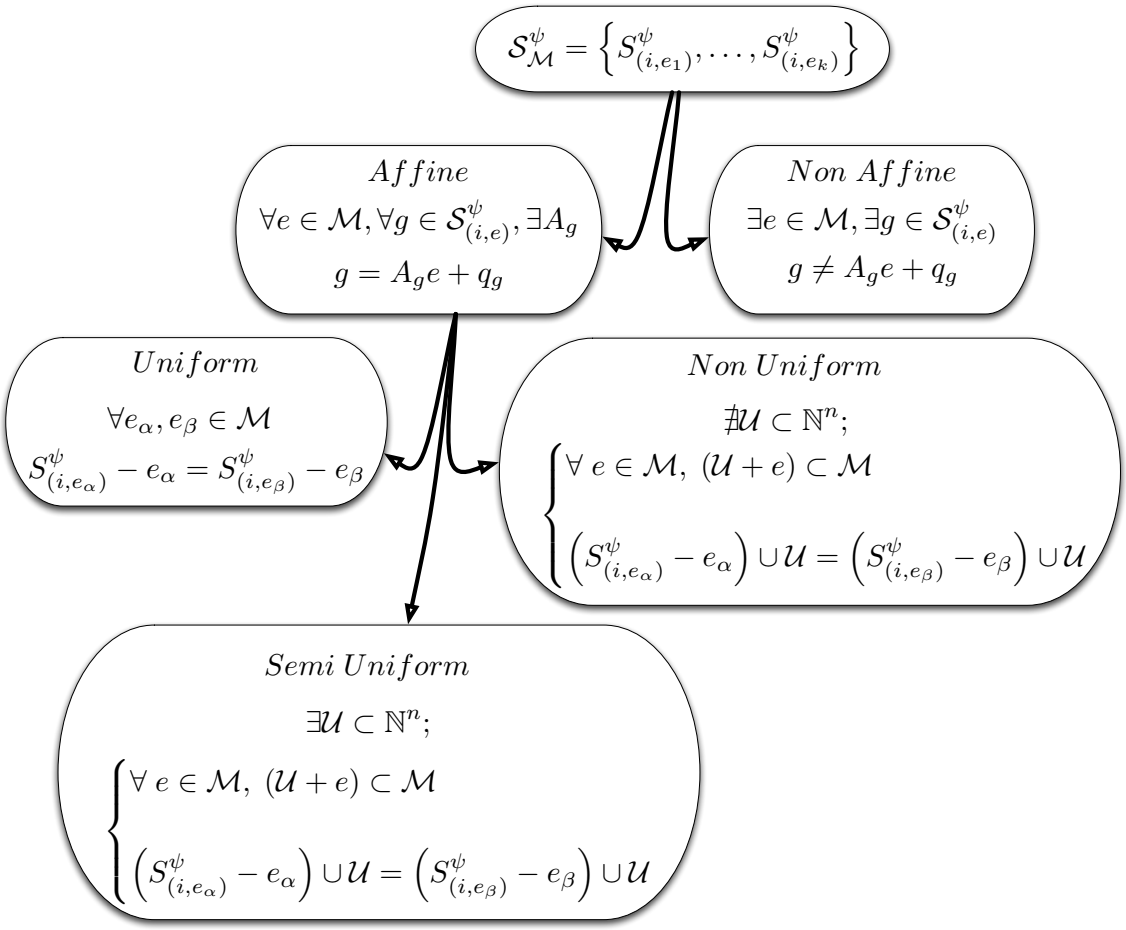


Figure 2.13: Classification of space invariant affine stencils

Another important structured model component, whose features can be used to catalogue stencils, is the shape set, which was previously defined for a generic stencil ψ in Definition 2.1.7 as:

$$\mathcal{S}^{\psi}_{\mathcal{M}} = \left\{\forall \mathcal{S}^{\psi}_{(i,e)} | e \in \mathcal{M}_{\psi}\right\}$$

The analysis of the shape set leads to different reflections, all linked to the aim of finding some shared structural characteristics between the shape set elements, which are in the form:

$$\mathcal{S}^{\psi}_{(i,e)} = \left\{g_1, g_2, \ldots, g_k | \ \forall \alpha\ g_\alpha \in \mathcal{M}_{\psi}\right\}$$

We recall that by definition there is one and only one element (i.e. shape) in the shape set that is associated with each spatial structure element (i.e. the application point). In turn, each shape set element is composed of a set of spatial structure elements.

In Figure 2.13, we give a preview of the stencil families that we will describe in the following Sections.

2.3.4 A Classification Based on Relations Between Application Point and Shape

We would now like to look at the characterization of stencils according to some relations between each shape element and the corresponding application point. In other words, we are trying to model those stencils whose shapes can be described parametrically with respect to their application points. With this specific aim, we define the following space invariant class.

Definition 2.3.5 (**Affine Space Invariant Stencil Class).** Let ψ be a space invariant stencil featuring $\mathcal{S}^{\psi}_{\mathcal{M}^i}$ as a parametric representation of the shape set in terms of its generic $step^{\psi}_i$. We define $step^{\psi}_i$ as an **affine step** if each of its shapes can be represented as a linear transformation of the corresponding application point:

$$\forall \mathcal{S}^{\psi}_{(i,e)} \in \mathcal{S}^{\psi}_{\mathcal{M}^i},\ \forall g \in \mathcal{S}^{\psi}_{(i,e)},\ \exists A_g,\ g = A_g e + q_g$$

A_g is the **rotation matrix**, which features a non zero determinant, while q_g is the **translation vector**.

We classify ψ as an **affine stencil** if all its steps are affine. A stencil that does not fall in this class is labelled as **non affine**.

The shape of an affine and space invariant stencil can therefore be represented as:

$$\mathcal{S}^{\psi}_{(i,e)} = \left\{ A_1 e + q_1, A_2 e + q_2, \ldots, A_k e + q_k | \ \forall \alpha \ (A_\alpha e + q_\alpha) \in \mathcal{M}_\psi \right\}$$

Consider for instance the stencil described by the following statement:

$$J_{in}(x, y) = J_{out}(x^2, y^2) + J_{out}(x, 3 * y) + J_{out}(x, y)$$

No one shape of the stencil can be defined as affine because each one has an element whose parametric description is not expressed as a linear function.

All the examples that we have presented up to now are classified as affine, and most of the real stencil applications belong to this class.

The affine classification is completely orthogonal to the homogeneous one. It is possible to have stencils which are homogeneous but not affine or vice versa; all the possible combinations are legal. This property of orthogonality comes from the fact that the two classifications focus on features which are associated with two independent stencil components: the step set for homogeneity and the shape elements for their descriptions in terms of linear transformations of the associated application point.

2.3.5 A Classification Based on the Relations Between Shapes

Considering the affine stencil set, we can study a finer classification analyzing not only the geometric features of a single shape but also the possible relations that can be highlighted between shapes of the same stencil step.

A significant relation is represented by a feature of the step set component: it can be composed only of equivalent shapes, as happens in both the Jacobi and Laplace examples. We therefore introduce this first sub-classification of affine stencil family.

Definition 2.3.6 (**Uniform Affine Stencil Class).** Let ψ be an affine stencil and let $\mathcal{S}^{\psi}_{\mathcal{M}^i}$ be its shape set for the step $step_i$. The step $step_i$ is classified as a **uniform step** if all its shapes are equivalent:

$$\forall e_\alpha, e_\beta \in \mathcal{M},\ \mathcal{S}^{\psi}_{(i,e_\alpha)} \triangleq \mathcal{S}^{\psi}_{(i,e_\beta)}$$

We classify ψ as a **uniform stencil** if all its steps are in turn uniform.

The definition introduces the uniform class directly as a subset of the affine one. The soundness of the definition is confirmed by the following theorem, which asserts that it is not possible for a uniform stencil to exist without belonging to the affine class.

Theorem 2.3.1 (**Uniform and Affine Classes Relation).** *The class of Uniform stencil is a proper subset of the Affine class:*

$$Affine \subset Uniform$$

Proof. The proof is quite easy and straightforward. If two shapes are equivalent, then we have:

$$\mathcal{S}^{\psi}_{(i,e_\alpha)} - e_\alpha = \mathcal{S}^{\psi}_{(i,e_\beta)} - e_\beta$$

Expanding the definition of the step set, we get:

$$\forall e_\alpha, e_\beta \in \mathcal{M},\ \left\{g^\alpha_1 - e_\alpha, g^\alpha_2 - e_\alpha, \ldots, g^\alpha_k - e_\alpha\right\} = \left\{g^\beta_1 - e_\beta, g^\beta_2 - e_\beta, \ldots, g^\beta_k - e_\beta\right\}$$

Recalling that the sets we are considering are ordered sets:

$$\forall e_\alpha, e_\beta \in \mathcal{M},\ g^\alpha_j - e_\alpha = g^\beta_j - e_\beta$$

The previous equality has to be valid for all the possible elements of the spatial structure, because of the definition of uniform stencil class. The only way to satisfy the equality is therefore that each element g_j is described parametrically with respect to the application point in the following way:

$$\mathcal{S}^{\psi}_{(i,e)} = \left\{\mathcal{I}e + q_1, \mathcal{I}e + q_2, \ldots, \mathcal{I} + q_k |\ \forall \alpha\ (\mathcal{I}e + q_\alpha) \in \mathcal{M}_\psi\right\}$$

where $\mathcal{I}$ is the identity matrix. The form of $\mathcal{S}^{\psi}_{(i,e)}$ implies that a uniform stencil is an affine stencil with the identity matrix as its rotation matrix. □

From the previous Theorem comes an important corollary on the analytic structure of shapes in semi-uniform stencils.

Corollary 2.3.1 (**Uniform Shape Structure**). *A necessary condition for a stencil to belong to the uniform class is that each shape is defined as follows:*

$$\mathcal{S}^{\psi}_{(i,e)} = \left\{ \mathcal{I}e + q_1, \mathcal{I}e + q_2, \ldots, \mathcal{I} + q_k | \; \forall \alpha \; (\mathcal{I}e + q_\alpha) \in \mathcal{M}_\psi \right\}$$

Where $\mathcal{I}$ is the identity matrix.

Proof. The proof comes directly from the demonstration of Theorem 2.3.1. □

Stencils belonging to the uniform class hide an important feature that strictly links them to toroidal spaces. This characteristic is introduced and proved in the following property:

Property 2.3.1 (**Uniform Stencil Toroidal Spaces**). *Uniform stencils are defined only over toroidal spaces.*

Proof. We use a *reductio ad absurdum* strategy for the demonstration; we therefore assume that there exists a uniform stencil ψ that is defined over a non toroidal regular spatial structure $\mathcal{M}^{\psi}$, characterized by a length vector $m = (m_1, \ldots, m_n)$.

By definition of uniformity and by Corollary 2.3.1, a uniform stencil features a set made up only of uniform steps, whose associated shape sets are in turn composed of all equivalent shapes, of the following form:

$$\mathcal{S}^{\psi}_{(i,e)} = \left\{ \mathcal{I}_1 e + q_1, \mathcal{I}_2 e + q_2, \ldots, \mathcal{I}_k e + q_k | \; \forall \alpha \; (\mathcal{I}_\alpha e + q_\alpha) \in \mathcal{M}_\psi \right\}$$

We suppose a vector element $q_\gamma = (q_0, \ldots, q_n)$ exists such that at least one component q_i is positive. Consider now the element of the spatial structure associated with the vector $e = (m_1 - 1, \ldots, m_n - 1)$ as application point. In the corresponding shape, there exists the element $e + q_\gamma = (m_0 + q_0, \ldots, m_i + q_i, \ldots, m_n + q_n)$, which does not belong to the spatial structure; indeed, its $i - th$ component, i.e. $m_i + qi$, is equal or greater than m_i. We can therefore claim that we are confronted by an *absurdum*, in fact by the definition of shape all the elements have to belong to the spatial structure of the stencil. The same reasoning can be applied symmetrically by considering a negative component of q_γ.

We can conclude that the only space where we can define a uniform stencil has to be toroidal. □

2.3.6 The Semi-Uniform Stencil Class

In this Section we cover the definition of another subclass of affine stencils. This subclass highlights a set of affine stencils which, with some extensions, can be easily attributed to the uniform class.

Consider as an example the Black step of the Red-Black application. If we analyze only the shapes associated with black elements, i.e. those featuring a cross geometry, we should say that the black shapes are affine and make up a uniform subset of the black shape set. The same reasoning can be applied symmetrically to red points, i.e. those featuring a shape that is composed of only one element. To summarize, all the red shapes and all the black ones can be expressed as affine transformations of the application point where the rotation matrix is the identity. This is a necessary, but not sufficient, condition for a stencil to be classified in the uniform stencil set as Corollary 2.3.1 claims.

What a black step lacks for it to be classified as uniform is having the same elements on black and red shapes. Nevertheless, this drawback can be avoided by defining an extension of the step such that the resulting step is uniform.

The key point is to extend, on a Black step, both the black and red shapes with a set of elements, in such a way that all the new shapes are equivalent. We call the set $\mathcal{U}$, the set of uniformity.

It is worth mentioning that the extension being considered is at the level of the structure of shapes, indeed the new added elements will not be considered in the the definition of the evaluation map.

We show the definition of $\mathcal{U}$ in the case of the Red-Black stencil in Table 2.7 for black steps and in Table 2.8 for red steps. Moreover, a graphical representation of the two extensions is reported respectively in Figure 2.15 and Figure 2.14.

Analyzing the set $\mathcal{S}^{RB}_{(x,y)} \cup \mathcal{U}$ as the new shape, we find that the extended versions of both black and red steps are classified as uniform, because all the spatial structure elements mow feature the same shape elements.

From the two examples, it becomes clear that, as previously noted, although the shapes were extended with new elements in order to reduce the stencil to an object of the uniform class, the step functions and evaluation map were not modified; they are still defined in terms of the elements of the original shapes.

Based on the previous observations, we introduce the following new sub-classification of affine stencils.

Definition 2.3.7 (**Semi-Uniform Stencil Class).** Let ψ be an affine stencil and $\mathcal{S}^{\psi}_{\mathcal{M}^i}$ its shape set for the step $step_i$. The step $step_i$ is classified as uniform if there exists a set $\mathcal{U} \subset \mathcal{N}^n$, called the **uniformity set**, such that:

$$\begin{cases} \forall e_\alpha, e_\beta \in \mathcal{M},\ \left(\mathcal{S}^{\psi}_{(i,e_\alpha)} \cup (\mathcal{U} + e_\alpha)\right) \stackrel{\diamondsuit}{=} \left(\mathcal{S}^{\psi}_{(i,e_\beta)} \cup (\mathcal{U} + e_\beta)\right) \\ \\ \forall\ e \in \mathcal{M}\ \ (\mathcal{U} + e) \subset \mathcal{M} \end{cases} \tag{2.5}$$

Stencil ψ is classified as a **semi-uniform stencil** if all its steps are semi-uniform; where each step can feature a different uniform set.

Informally, we say that a stencil belongs to the semi-uniform class if we can define a set, parametric with respect to the application point, in such a way that

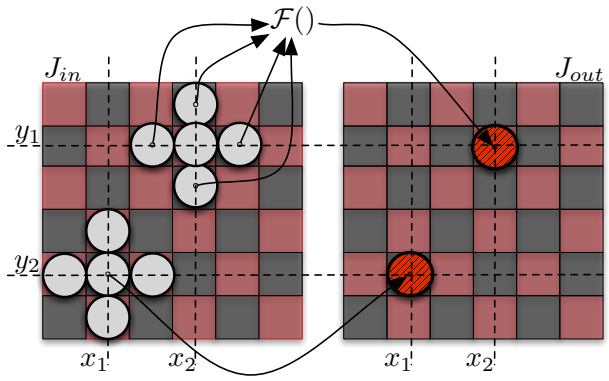


Figure 2.14: Representation of two application points and the corresponding shapes which are featured by the Red step of the extended version of the Black-Red application whose description in pseudo-code is reported in Figure 2.7. The diagram can be compared with the original one reported in Figure 2.8

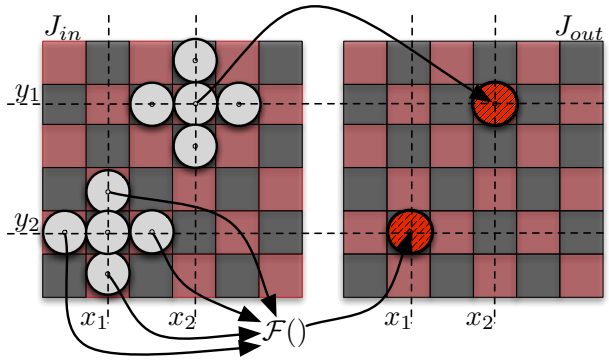


Figure 2.15: Representation of two application points and the corresponding shapes which are featured by the Black step of the extended version of the Black-Red application whose description in pseudo-code is reported in Figure 2.7. The diagram can be compared with the original one reported in Figure 2.9.

Extended 2D Red step Model

$$\forall e=(x,y)\in\mathcal{M}_{RB} \overset{black_i}{\rightarrow} \left(\mathcal{F}_{ie},\mathcal{S}^{RB}_{ie}\right)$$

$$\mathcal{U} = \big\{(0,0),(0,+1),(0,-1),(+1,0),(-1,0)\big\}$$

$$\mathcal{S}^{RB}_{(x,y)}\cup(\mathcal{U}+e) = \begin{cases} \big\{(x,y+1),(x,y-1),(x+1,y),(x-1,y)\big\}\cup \\ \cup\overbrace{\big\{(x,y),(x,y+1),(x,y-1),(x+1,y),(x-1,y)\big\}}^{\mathcal{U}+e} \\ \textbf{if } e \text{ is black} \\ \\ \big\{(x,y)\big\}\cup \\ \cup\overbrace{\big\{(x,y),(x,y+1),(x,y-1),(x+1,y),(x-1,y)\big\}}^{\mathcal{U}+e} \\ \textbf{if } e \text{ is red} \end{cases}$$

$$\mathcal{F}_{(i,e)} : \begin{cases} \mathbb{R}^4\mapsto\mathbb{R} \text{ if } e \text{ is black} \\ \\ \mathbb{R}^1\mapsto\mathbb{R} \text{ if } e \text{ is red} \end{cases}$$

$$\mathcal{M}^{i+1}_{RB}[e] = \begin{cases} \frac{1}{4}\big(\mathcal{M}^i_{RB}[(x,y+1)]+\mathcal{M}^i_{RB}[(x,y-1)] \\ +\mathcal{M}^i_{RB}[(x+1,y)]+\mathcal{M}^i_{RB}[(x-1,y)]\big) \text{ if } e \text{ is red} \\ \\ \mathcal{M}^i_{RB}[e] \textbf{ if } e \text{ is black} \end{cases}$$

Table 2.7: Structured Step model of the Red steps of the **extended** Red-Black (RB) stencil application described in Figure 2.7

Extended 2D Black step Model

$$\forall e=(x,y)\in\mathcal{M}_{RB} \overset{red_i}{\rightarrow} \left(\mathcal{F}_{ie},\mathcal{S}^{RB}_{ie}\right)$$

$$\mathcal{U} = \big\{(0,0),(0,+1),(0,-1),(+1,0),(-1,0)\big\}$$

$$\mathcal{S}^{RB}_{(x,y)}\cup(\mathcal{U}+e) = \begin{cases} \big\{(x,y+1),(x,y-1),(x+1,y),(x-1,y)\big\}\cup \\ \cup\overbrace{\big\{(x,y),(x,y+1),(x,y-1),(x+1,y),(x-1,y)\big\}}^{\mathcal{U}+e} \\ \textbf{if } e \text{ is black} \\ \\ \big\{(x,y)\big\}\cup \\ \cup\overbrace{\big\{(x,y),(x,y+1),(x,y-1),(x+1,y),(x-1,y)\big\}}^{\mathcal{U}+e} \\ \textbf{if } e \text{ is red} \end{cases}$$

$$\mathcal{F}_{(i,e)} : \begin{cases} \mathbb{R}^4\mapsto\mathbb{R} \textbf{ if } e \text{ is red} \\ \\ \mathbb{R}^1\mapsto\mathbb{R} \textbf{ if } e \text{ is black} \end{cases}$$

$$\mathcal{M}^{i+1}_{RB}[e] = \begin{cases} \frac{1}{4}\big(\mathcal{M}^i_{RB}[(x,y+1)]+\mathcal{M}^i_{RB}[(x,y-1)] \\ +\mathcal{M}^i_{RB}[(x+1,y)]+\mathcal{M}^i_{RB}[(x-1,y)]\big) \textbf{ if } e \text{ is black} \\ \\ \mathcal{M}^i_{RB}[e] \textbf{ if } e \text{ is red} \end{cases}$$

Table 2.8: Structured Step model of the Black steps of the **extended** Red-Black (RB) stencil application described in Figure 2.7

its elements, added to each shape of the step, transform the extended stencil into a uniform one. There is a condition to respect in the definition of $\mathcal{U}$. The elements that are added to a shape, i.e. the set $\mathcal{U} + e$, have to form a strict subset of the spatial structure $\mathcal{M}$: $(\mathcal{U} + e) \subset \mathcal{M}$.

From the definition we can derive a necessary condition in order to classify a stencil as semi-uniform.

Theorem 2.3.2 (**Semi-Uniform Stencil Shape Structure**). *A necessary condition for a stencil step to belong to the semi-uniform class is that each of its shapes* $\mathcal{S}^{\psi}_{(i,e)}$ *is of the form:*

$$\mathcal{S}^{\psi}_{(i,e)} = \left\{\mathcal{I}e + q_1, \mathcal{I}e + q_2, \ldots, \mathcal{I} + q_k | \ \forall \alpha \ (\mathcal{I}e + q_\alpha) \in \mathcal{M}_\psi\right\}$$

where $\mathcal{I}$ *is the identity matrix.*

Proof. From Corollary 2.3.1, we know that in a uniform step all the shapes can be modelled in the following form:

$$\mathcal{S}^{\psi}_{(i,e)} = \left\{\mathcal{I}e + q_1, \mathcal{I}e + q_2, \ldots, \mathcal{I} + q_k | \ \forall \alpha \ (\mathcal{I}e + q_\alpha) \in \mathcal{M}_\psi\right\}$$

Thus, in the case of a semi-affine stencil we have that: $\forall e_\alpha, e_\beta \in \mathcal{M}$,

$$\left(\mathcal{S}^{\psi}_{(i,e_\alpha)} \cup (\mathcal{U} + e_\alpha)\right) = \left\{\mathcal{I}e + q_1, \mathcal{I}e + q_2, \ldots, \mathcal{I} + q_k | \ \forall \alpha \ (\mathcal{I}e + q_\alpha) \in \mathcal{M}_\psi\right\}$$

Each element of the two components of the union operation has to be defined as $\mathcal{I}e_\alpha + q_\alpha$ □

Turning to the uniformity sets, we are interested in defining the smallest one, i.e. the one featuring the lowest cardinality. Indeed, if the smallest one is not a subset of $\mathcal{M}$, we can claim that the associated stencil cannot be classified as semi-uniform. Therefore, we present the following Theorem:

Theorem 2.3.3 (**The Smallest Uniformity Set**). *Let* ψ *be a semi uniform stencil and* $\mathcal{S}^{\psi}_{\mathcal{M}^i}$ *its shape set for the step* $step_i$*. The smallest uniformity set* $\mathcal{U}_i^{MIN}$ *for* $step_i$ *is given by the following formula:*

$$\mathcal{U}_i^{MIN} = \bigcup_{\forall e \in \mathcal{M}} \left(\mathcal{S}^{\psi}_{(i,e)} - e\right) \tag{2.6}$$

Proof. The demonstration is quite trivial. Firstly, $\mathcal{U}_i^{MIN}$ is a uniformity set for $step_i$. Indeed according to Definition 2.3.3 of shape equivalence, we know that Equation 2.8 can be rewritten as:

$$\forall e_\alpha, e_\beta \in \mathcal{M}, \ \left(\mathcal{S}^{\psi}_{(i,e_\alpha)} - e_\alpha\right) \cup \mathcal{U}_i^{MIN} = \left(\mathcal{S}^{\psi}_{(i,e_\beta)} - e_\beta\right) \cup \mathcal{U}_i^{MIN}$$

The previous equation is guaranteed by the construction of $\mathcal{U}_i^{MIN}$.

Secondly, by *absurdum*, we suppose that there exists $\overline{\mathcal{U}}$, a smaller uniformity set than $\mathcal{U}_i^{MIN}$. Because the new set is smaller, there exists an element g such that $g \in \mathcal{U}_i^{MIN}$ and $q \notin \overline{\mathcal{U}}$. By construction of $\mathcal{U}_i^{MIN}$ there exists at least one spatial structure element e_q such that its shape contains $e_q + q$:

$$(e_q + q) \in \mathcal{S}^{\psi}_{(i,e_\alpha)}$$

Because $e_q + q$ belongs to a step shape but not to $\overline{\mathcal{U}}$, this last set cannot be a uniformity set and we can assert that we have reached a contradiction. We therefore conclude that $\mathcal{U}_i^{MIN}$ is the smallest uniformity set. □

Concerning the definition of the semi-uniform stencil class, it is worth mentioning that $\mathcal{U}$ is required to be a strict subset of $\mathcal{M}^{\psi}$. Consider the case of Floyd-Warshall: each shape has different elements, but, in contrast to the Red-Black stencil, the shapes depend on the step index which characterizes typical triangular access to the matrix. In such a case, the only uniform set is the entire spatial structure $\mathcal{M}$, therefore the Floyd-Warshall stencil is not classified as semi-uniform.

Theorem 2.3.4 (**Semi-Uniformity**). *A sufficient and necessary condition for a step $step_i$ to be classified as semi-uniform is validity of the following formula:*

$$\forall e \in \mathcal{M},\ \mathcal{U}_i^{MIN} \subset \mathcal{M} \tag{2.7}$$

Proof. The proof comes directly from Definition 2.3.8 and Theorem 2.3.3 □

From the previous Theorem we can conclude that semi-affinity is a characteristic that can be automatically proved knowing all the information about the steps of the stencil.

2.3.7 The Space Invariant Stencil Family: The Big Picture

In this Section, we would like to summarize all the properties of the space invariant stencils that we presented in the previous Sections.

The relations between the different stencil classes are straightforward to present. First of all, we have two broad families of stencils that represent Homogeneous and Affine classes. As previously noted, the two classes are orthogonal to one another, i.e. they are independent: an homogeneous stencil can be affine or not and vice versa. While the homogeneous class does not have any other sub-classification, in the affine set we distinguish three sub-classes: non-uniform, semi-uniform and uniform. Until now, we have only demonstrated in Proposition 2.3.1 that uniform stencils are completely contained in the affine class and we also know by definition that a non-uniform stencil does not contain either uniform or semi-uniform stencils. What remains to model is the correspondence between uniform and semi-uniform, we therefore introduce the following property.

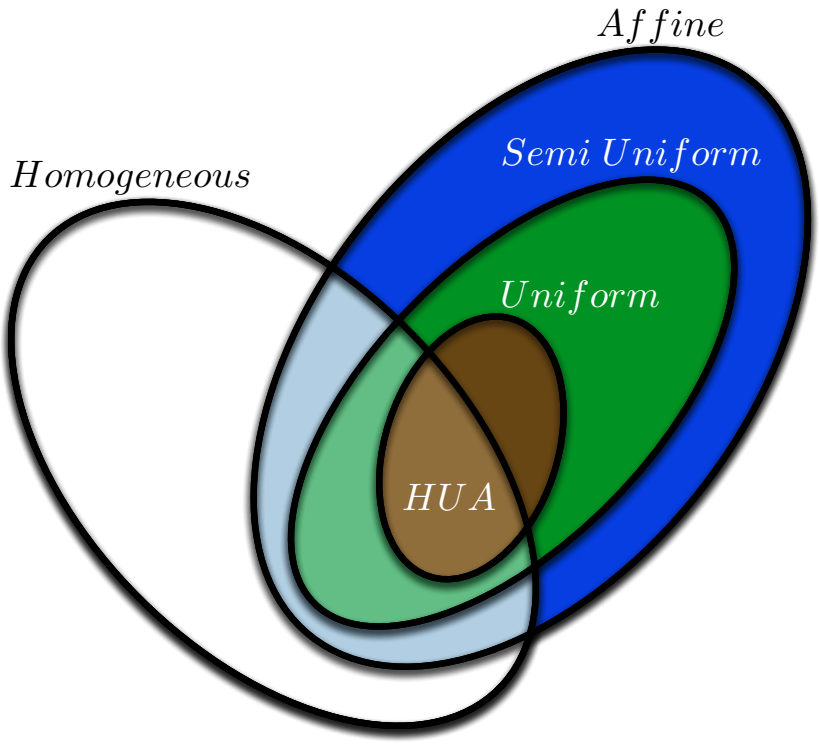


Figure 2.16: Representation of HUA space invariant stencil compared with all presented stencil classifications.

Property 2.3.2 (Uniform and Semi-Uniform Classes Relations). *The uniform class is a proper subclass of the semi-uniform one:*

$$Uniform \subset Semi\ Uniform$$

Proof. Exploiting Theorem 2.3.4, it is trivial to see that for a uniform stencil we have:

$$\bigcup_{\forall e \in \mathcal{M}} \left(\mathcal{S}^{\psi}_{(i,e)} - e \right) = \bigcap_{\forall e \in \mathcal{M}} \left(\mathcal{S}^{\psi}_{(i,e)} - e \right) \subset \mathcal{M}$$

□

Now that we have completed the entire set of relations between the different classes of space invariant stencils, we can summarize all the information already acquired in the representation of Figure 2.16.

An important aspect of the graphical representation is the intersection of uniform and homogeneous stencils: we call this class $\mathcal{HUA}$, which stands for **Homogeneous**,

Uniform and **Affine**. In the rest of the thesis this will be our main topic. All the transformations we are going study are first presented for $\mathcal{HUA}$ stencils and then extended to other classes. Therefore, because of the relevance of the $\mathcal{HUA}$ stencil to our study, in Section 2.5 we will introduce a specification of the structured model that explicitly highlights, with proper mechanisms, the distinguishing features of this stencil class.

2.3.8 Boundary Problems

```
                                                                    1
double J_in[10], J_out[10];                                         2
load_working_domain_values(J_in);                                   3
for(i_step = 0; i_step < 4; i_step++){                              4
    forall(x ∈ J_in){                                               5
        if(x is left border){                                       6
            J_out[x] = J_in[x + 1]                                  7
        }                                                           8
                                                                    9
        if(x is not border){                                        10
            J_out[x] = 1/2(J_in[x] + J_in[x + 1] + J_in[x − 1]      11
        }                                                           12
                                                                    13
        if(x is right border){                                      14
            J_out[x] = J_in[x − 1]                                  15
        }                                                           16
    }                                                               17
    swap(J_in, J_out);                                              18
                                                                    19
}                                                                   20
return_working_domain(J_out);                                       21
```

Figure 2.17: Representation in pseudo-code of a Jacobi stencil application in one-dimensional non toroidal space.

All the examples that we have presented up to now, except for Floyd-Warshall, were defined over toroidal spaces, but usually, in real world examples, the application spaces are not toroidal. In this case, the problem of managing the spatial structure bounds arises.

We have seen with the proof of Theorem 2.3.1 that a uniform stencil implies a toroidal space. In stencils defined over non toroidal space, the stencil shape which is

associated with elements next to the space bounds, have to be managed differently from others, i.e. applications with bound space do not belong to the uniform class.

We consider the case of the one-dimensional Jacobi defined by a regular spatial structure, which by definition is not toroidal, and we analyze what kind of information we can extract by analyzing its structured model. We choose to present a one-dimensional space example instead of reusing the two-dimensional Jacobi, for the sake of simplicity: in a one-dimensional space we face only two classes of bound elements, while in a two-dimensional case there are eight.

We report the pseudo-code in Figure 2.17 and a graphical representation in Figure 2.19. The non toroidal algorithm features, inside the *forall* construct, three *if* statements in order to divide the spatial structure elements in the following regions: *left border*, *right border* and *internal*. The classical one-dimensional Jacobi shape is associated with elements of the *internal* region, in the same way it would be in the case of a toroidal structure. Instead, to the elements of the *left border* and *right border* regions, which in this particular case are composed of one element each, a reduced shape is associated, i.e. a classical shape thinned out of those elements that would not be inside the spatial structure.

We report the structured model of the one-dimensional Jacobi in Table 2.9. In the description, it is interesting to highlight two main aspects. Firstly, the working domain is defined over a non toroidal domain. Secondly, both the shape and step function declarations are described parametrically with respect to the membership of a spatial structure element to one of the three regions.

It is now interesting to consider the classification of the one-dimensional Jacobi. Firstly, the stencil application belongs indisputably to the intersection of both homogeneous and affine classes. Secondly, the stencil is undeniably from the uniform class, because the shapes associated with the borders are not equivalent to each other and in turn neither is equivalent to the shape of the internal region elements. Considering the big picture of stencil classification (see Figure 2.16), it remains an open problem whether the stencil should be classified as semi-uniform or not.

Following the some path as for the Red-Black stencil, which took us to the definition of the semi-uniform class, we study an extension of the Jacobi model which is reported in Table 2.10 and is represented graphically in Figure 2.19.

The extended model is indisputably uniform, but we were able to define the set $\mathcal{U}$ only by paying the price of remapping the extended model on a toroidal space: $\mathcal{W}_{JCB} = \left(\frac{\mathbb{Z}}{10}, \mathbb{R}, \mathcal{M}^i_{JCB}[.]\right)$. Indeed, a set of uniformity does not exist in which the extended shape of the left and right border elements are equivalent to the extended shape of the internal section elements. The previous claim is guaranteed by the necessary condition for semi-uniformity claimed by Theorem 2.3.4.

To summarise, we have defined a Jacobi stencil extension that is classified as uniform. Nevertheless, in contrast with the case of the Red-Black application, because of the existence of the uniform extended version, we cannot classify, according to the current model formalization, the original non toroidal Jacobi as semi-uniform. Indeed, the definition of semi-uniform class (see Definition 2.3.8) establishes that a

stencil can be classified as semi-uniform if there exists an extension, modelled by a set named the uniformity set, which is classified as uniform. It is worth mentioning that the definition does not consider that the extension, as happened for the Jacobi case, is provided along with a remapping of the spatial structure from regular to toroidal. We therefore introduce a specification for the semi-uniformity definition of regular structures as follows

Definition 2.3.8 (**Extension of the Semi-Uniform Stencil Class for Regular Structures).** Let ψ be an affine stencil that is defined over a non toroidal spatial structure and let $\mathcal{S}^{\psi}_{\mathcal{M}^i}$ be its shape set for the step $step_i$. The step $step_i$ is classified as semi-uniform if, after a remapping of the spatial structure onto a toroidal space, there exists a set $\mathcal{U} \subset \mathcal{N}^n$, called the **uniformity set**, such that:

$$\begin{cases} \forall e_\alpha, e_\beta \in \mathcal{M},\ \left(\mathcal{S}^{\psi}_{(i,e_\alpha)} \cup (\mathcal{U} + e_\alpha)\right) \triangleq \left(\mathcal{S}^{\psi}_{(i,e_\beta)} \cup (\mathcal{U} + e_\beta)\right) \\ \\ \forall\ e \in \mathcal{M}\ \ (\mathcal{U} + e) \subset \mathcal{M} \end{cases} \tag{2.8}$$

Stencil ψ is classified as a **semi-uniform stencil** if all its steps are semi-uniform; where each step can feature a different uniform set.

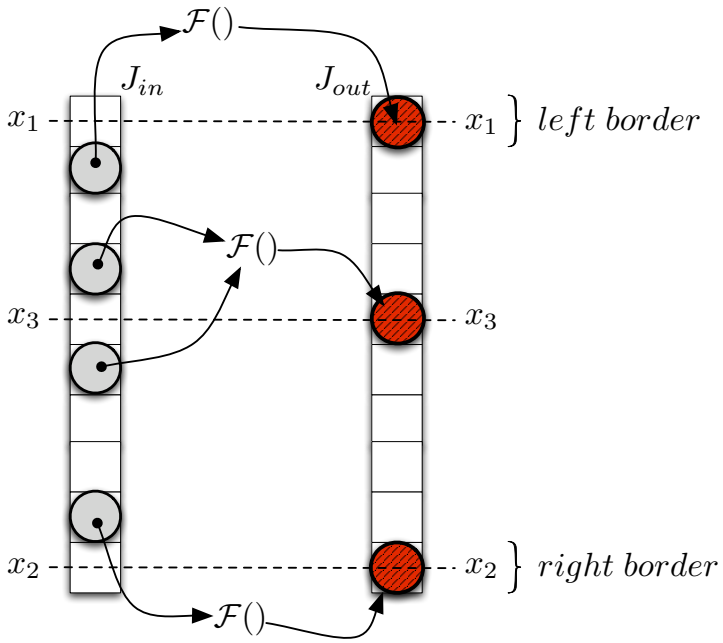


Figure 2.18: Representation of some application points and the corresponding shapes which are featured by the one-dimensional non toroidal Jacobi application whose description in pseudo-code is reported in Figure 2.17.

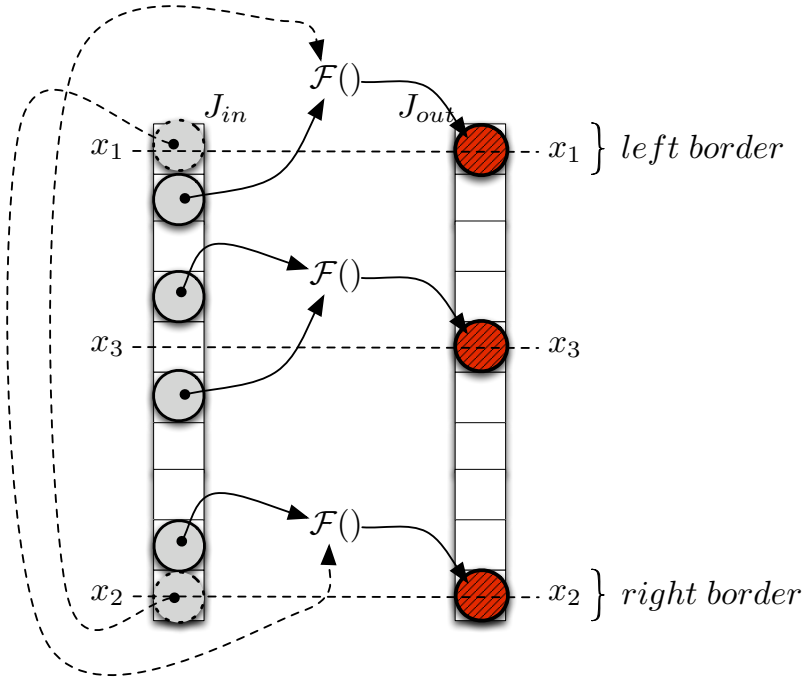


Figure 2.19: Representation of some application points and the corresponding shapes which are featured by the **extended** one-dimensional non toroidal Jacobi application whose description in pseudo-code is reported in Figure 2.17.

1D NON toroidal Jacobi (JCB) Step Model

$$
\begin{aligned}
JCB &= \big(\mathcal{W}_{JCB}, \mathcal{T}_{JCB}\big) \\
\mathcal{W}_{JCB} &= \big([0,9], \mathbb{R}, \mathcal{M}^i_{JCB}[.]\big) \\
m_{JCB} &= (10) \\
\mathcal{T}_{JCB} &= \big\{step_1^{JCB}, step_2^{JCB}, step_3^{JCB}, step_4^{JCB}\big\} \\
\mathcal{C}_{JCB} &= [1,5]
\end{aligned}
$$

$$
\forall e = x \in \mathcal{M}_{JCB} \overset{step_i}{\rightarrow} \big(\mathcal{F}_{ie}, \mathcal{S}^{JCB}_{ie}\big)
$$

$$
\mathcal{S}^{JCB}_{(x,y)} = \begin{cases} \big\{(x+1)\big\} \textbf{ if } e \text{ is left border} \\ \big\{(x-1),(x+1)\big\} \textbf{ if } e \text{ is not border} \\ \big\{(x-1)\big\} \textbf{ if } e \text{ is right border} \end{cases}
$$

$$
\mathcal{F}_{(i,e)} : \begin{cases} \mathbb{R} \mapsto \mathbb{R} \textbf{ if } e \text{ is left border} \\ \mathbb{R}^2 \mapsto \mathbb{R} \textbf{ if } e \text{ is not border} \\ \mathbb{R} \mapsto \mathbb{R} \textbf{ if } e \text{ is right border} \end{cases}
$$

$$
\mathcal{M}^{i+1}_{JCB}[e] = \begin{cases} \mathcal{M}^i_{JCB}[(x+1)] \textbf{ if } e \text{ is left border} \\ \frac{1}{2}\big(\mathcal{M}^i_{JCB}[(x+1)] + \mathcal{M}^i_{JCB}[(x-1)]\big) \textbf{ if } e \text{ is not border} \\ \mathcal{M}^i_{JCB}[(x-1)] \textbf{ if } e \text{ is right border} \end{cases}
$$

Table 2.9: Structured Step model of the 1D non toroidal Jacobi stencil (JCB) described in the pseudo-code of Figure 2.17.

1D NON toroidal Jacobi (JCB) Step Model

$$
\begin{aligned}
JCB &= \left(\mathcal{W}_{JCB}, \mathcal{T}_{JCB}\right) \\
\mathcal{W}_{JCB} &= \left(\frac{\mathbb{Z}}{10}, \mathbb{R}, \mathcal{M}^i_{JCB}[.]\right) \\
m_{JCB} &= (10) \\
\mathcal{T}_{JCB} &= \left\{step_1^{JCB}, step_2^{JCB}, step_3^{JCB}, step_4^{JCB}\right\} \\
\mathcal{C}_{JCB} &= [1,5]
\end{aligned}
$$

$$
\begin{aligned}
\forall e = x \in \mathcal{M}_{JCB} &\overset{step_i}{\rightarrow} \left(\mathcal{F}_{ie}, \mathcal{S}^{JCB}_{ie}\right) \\
\mathcal{U} &= \left\{+1, -1\right\} \\
\mathcal{S}^{JCB}_{(x,y)} \cup (\mathcal{U}+e) &= \begin{cases}
\left\{(x+1)\right\} \cup \overbrace{\left\{(x+1),(x-1)\right\}}^{\mathcal{U}+e} & \textbf{if } e \text{ is left border} \\
\left\{(x-1),(x+1)\right\} \cup \overbrace{\left\{(x+1),(x-1)\right\}}^{\mathcal{U}+e} & \textbf{if } e \text{ is not border} \\
\left\{(x-1)\right\} \cup \overbrace{\left\{(x+1),(x-1)\right\}}^{\mathcal{U}+e} & \textbf{if } e \text{ is right border}
\end{cases} \\
\mathcal{F}_{(i,e)} &: \begin{cases}
\mathbb{R} \mapsto \mathbb{R} & \textbf{if } e \text{ is left border} \\
\mathbb{R}^2 \mapsto \mathbb{R} & \textbf{if } e \text{ is not border} \\
\mathbb{R} \mapsto \mathbb{R} & \textbf{if } e \text{ is right border}
\end{cases} \\
\mathcal{M}^{i+1}_{JCB}[e] &= \begin{cases}
\mathcal{M}^i_{JCB}[(x+1)] & \textbf{if } e \text{ is left border} \\
\frac{1}{2}\left(\mathcal{M}^i_{JCB}[(x+1)] + \mathcal{M}^i_{JCB}[(x-1)]\right) & \textbf{if } e \text{ is not border} \\
\mathcal{M}^i_{JCB}[(x-1)] & \textbf{if } e \text{ is right border}
\end{cases}
\end{aligned}
$$

Table 2.10: Structured Step model of the extended 1D non toroidal Jacobi stencil (JCB) described in the pseudo-code of Figure 2.17

2.4 An Extension of Space Invariant Stencils

It is beyond the scope of this Section to discuss, in an informal way, some extensions to model non space invariant stencils. We recall that we defined the space invariant stencils as ones that feature one input and one output data structure that share the same index space. We showed that some stencil applications like the Jacobi, Laplace, Red-Black and Floyd-Warshall stencils belong to the space invariant stencil class.

We focus now on another application example, which is based on a reduce stencil. We consider a simple reduce operation that sums all the elements in a vector as shown in Figure 2.20.

In a generic $step_i$, the index space of the input data structure is wider than the index space of the output structure. Moreover, when we study the step set $\mathcal{T} = \{step_1, step_2, step_3\}$ we find that no single step features the same input index space. In order to model this kind of stencil, an extension of the model is required.

In order to manage the reduce stencil example, the model must therefore consider a new definition of the working domain. As for the step set component, the working domain has to be redefined as a container of *step working domains*, with one domain for each step.

For instance, in the case of the reduce example, we would have a working domain description such as the following:

$$\mathcal{W}_{reduce} = \left\{\mathcal{W}_1^{reduce}, \mathcal{W}_2^{reduce}, \mathcal{W}_3^{reduce}\right\}$$

Concerning each step working domain component, it can be described with a single parametric representation relative to the associated step index i_{step}:

$$\mathcal{W}_{i_{step}}^{reduce} = \left(\mathcal{M}_{i_{step}}^{IN} = \left[0, \left(\frac{8}{2^{i_{step}-1}} - 1\right)\right], \mathcal{M}_{i_{step}}^{OUT} = \left[0, \left(\frac{8}{2^{i_{step}}} - 1\right)\right], \mathbb{R}, \mathcal{M}_{reduce}^{i_{step}}[.]\right)$$

The re-definition of the working domain and especially the introduction in each single step working domain of two spatial structures, one for the input and one for the output, has a large impact on the step stencil component. The main change is reflected in the fact that the shape and application point are now defined over two different index spaces. The generic shape groups elements of the input structure while the application points are identified as elements of the output data structure. A possible step model for the reduce application is reported in Table 2.11. In the representation, we make use of the following re-definition of the evaluation map:

$$\mathcal{M}_{reduce}^{i}[e] = \mathcal{M}_i^{IN}[e], \ \forall \ e \ \in \mathcal{M}_i^{IN}$$

It is worth mentioning that, from the definition of the steps of the reduced application, the shape description is independent of the step it belongs to, and moreover it is expressed as a linear transformation of the application point. Therefore, the reduce application should be classified as a non space invariant and affine stencil.

The reduce application is just one example of a non space invariant stencil. For instance we could mention the matrix multiplication algorithm. In this case, which is different from the reduce example, we have to model more than one input structure in the working domain: one for each of the two input matrices.

Because in the rest of the thesis we concentrate on space invariant stencils, we consider what we have presented on non space invariant stencil to be a sufficient overview. We leave the formal description of an extended model, which can manage non space invariant stencils, to future works.

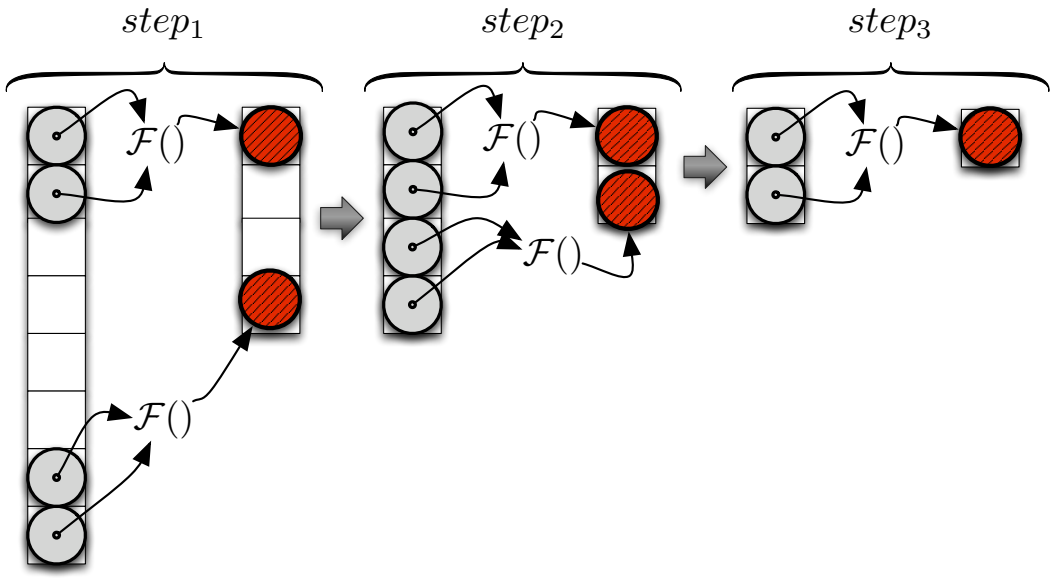


Figure 2.20: Representation of some application points and the corresponding shapes which are featured, during different steps, by a reduce application.

REDUCE STEP MODEL

$$\forall e = x \in \mathcal{M}_i^{OUT} \overset{step_i^{reduce}}{\rightarrow} \left(\mathcal{F}_{ie}, \mathcal{S}_{ie}^{JCB}\right)$$

$$\mathcal{S}_x^{JCB} = \left\{\forall\, \overline{e} \in \mathcal{M}_i^{IN} |\, \overline{e} \in \{x, 2x\}\right\}$$

$$\begin{aligned} \mathcal{F}_{(i,e)} &: \mathbb{R}^2 \mapsto \mathbb{R} \\ \mathcal{F}_{(i,e)} &: \left(r_1, r_2\right) \mapsto r_1 + r_2 \; \forall r_i \in \mathbb{R} \end{aligned}$$

$$\begin{aligned} \mathcal{M}_{JCB}^{i+1}[e] &= \mathcal{F}_{(i,e)}\left(\mathcal{M}_{reduce}^{i}[x], \mathcal{M}_{reduce}^{i}[2x]\right) \\ &= \mathcal{F}_{(i,e)}\left(\mathcal{M}_{i}^{IN}[x], \mathcal{M}_{i}^{IN}[2x]\right) \\ &= \mathcal{M}_{i}^{IN}[x] + \mathcal{M}_{i}^{IN}[2x] \end{aligned}$$

Table 2.11: Step component of a reduce application in the structured stencil model.

2.5 A Specification of the Structured Model for $\mathcal{HUA}$ Stencils

In the previous Sections, we have covered the formal definition of the model and an in depth classification of space invariant stencils. Now we would like to focus the discussion on a particular class called $\mathcal{HUA}$. This class comes from the intersection of the homogeneous and uniform classes. Because in the thesis the $\mathcal{HUA}$ stencils will play a fundamental role, we present a specialization of the structured model for $\mathcal{HUA}$ stencils, in order to better manage their distinguishing features.

2.5.1 Definition of the $\mathcal{HUA}$ Model

Before we start a formal introduction of the $\mathcal{HUA}$ model, it is worth indicating the features of the $\mathcal{HUA}$ stencil class which we wish to explicitly represent.

I A $\mathcal{HUA}$ stencil belongs to the homogeneous class, thus its step set is composed only of equivalent elements.

II A generic $\mathcal{HUA}$ stencil belongs also to the affine class, therefore each element of a shape can be represented as an affine transformation of the associated application point.

III Because a $\mathcal{HUA}$ stencil is classified as both uniform and homogeneous, we can claim that the same shape is independently associated to each element of the spatial structure and moreover the association is an invariant for all stencil steps.

IV Because a $\mathcal{HUA}$ stencil belongs to the uniform class, that is a subclass of the affine family, we can assert that the rotation matrix of all the affine transformations which are exploited to define the shape elements, is the identity matrix (see Corollary 2.3.1).

V Finally, from Property 2.3.1, we know that a $\mathcal{HUA}$ stencil can be defined only over toroidal spatial structures.

We can now give the following formal definition:

Definition 2.5.1 (**Homogeneous Uniform Affine Model).** Let ψ be a space invariant stencil belonging to the intersection of the homogeneous and uniform classes.

The *HUA* model represents one of the **homogeneous** steps of ψ as follows:

$$\begin{array}{rcl} \forall e \in \mathcal{M} & \stackrel{step_i}{\rightarrow} & (\mathcal{F}, \mathcal{S}) \\ \mathcal{S} & = & \{g_1, g_e, \ldots, g_n | \ \forall \alpha \ g_\alpha = (\mathcal{I}e + \beta_\alpha) \in \mathcal{M}\} \\ & \Downarrow & \\ & \triangleq & e + \{\beta_1, \beta_2, \ldots, \beta_n\} \\ & \Downarrow & \\ & \triangleq & e + \mathcal{R} \\ \mathcal{M}^{i+1}[e] & = & \mathcal{F}(\mathcal{M}^i[g_1], \mathcal{M}^i[g_2], \ldots, \mathcal{M}^i[g_n]) \\ & \Downarrow & \\ & \triangleq & \mathcal{F}(\mathcal{M}^i[e + \beta_1], \ldots, \mathcal{M}^i[e + \beta_n]) \end{array} \tag{2.9}$$

The matrix $\mathcal{I}$ is the identity matrix while the set $\mathcal{R}_i = \{\beta_1, \ldots \beta_n\}$ is called the **relative shape**.

In $\mathcal{HUA}$ both the same function $\mathcal{F}$ and the same relative shape $\mathcal{R}$ are **uniformly** associated to all the domain elements independently of the considered step.

Because for uniform stencils the rotation matrix is equal to the identity matrix, we can claim that $\mathcal{R}$ represents the minimum unit of information required to completely define the affine transformations of all the shape elements. The relative shape is the distinguishing contribution that is added by the $\mathcal{HUA}$ model to the more general structured one. The relative shape will be a tool that facilitates the proof of $\mathcal{HUA}$ stencil proprieties.

Previously in the Chapter, we analyzed two applications which belong to the $\mathcal{HUA}$ class: Jacobi and Laplace. We now would like to consider once again the Laplace application as an example of modelling with the new $\mathcal{HUA}$ formalism. The Laplace $\mathcal{HUA}$ model is reported in Table 2.12 and it can be compared with the general one reported in Table 2.2 in Section 2.2. The new representation stresses the fact that all the steps are equivalent, that the step function does not depend on any specific domain element. Finally and foremost, the model represents a new perspective on the shape component: a rigid translation of the relative shape on the application point.

2.5.2 Relaxed Computational Equivalence in the $\mathcal{HUA}$ Model

In order to give a preview of how the relative shape can be exploited to prove $\mathcal{HUA}$ stencil properties, we go back to the Introduction Chapter to recall the comparison of the mono-dimensional Jacobi to the modified version. We report in Figure 2.21 the graphical representation of the shapes and application points of both the original and modified Jacobi versions. We highlight once again that the only difference between the two stencils is given by the location of the application point relative to the shape (compare Figure 2.21(a) and Figure 2.21(b)).

2D Laplace (LPC) $\mathcal{HUA}$ Model

$$
\begin{aligned}
LPC &= \left(\mathcal{W}_{LPC}, \mathcal{T}_{LPC}\right) \\
\mathcal{W}_{LPC} &= \left(\frac{\mathbb{Z}}{1024} \times \frac{\mathbb{Z}}{1024}, \mathbb{R}, \mathcal{M}^{i}_{LPC}[.]\right) \\
m_{LPC} &= (1024, 1024) \\
\mathcal{T}_{LPC} &= \left\{step^{LPC}, step^{LPC}, step^{LPC}, step^{LPC}\right\} \\
\mathcal{C}_{LPC} &= [1, 5]
\end{aligned}
$$

$$
\begin{aligned}
\forall e = (x, y) \in \mathcal{M}_{LPC} \overset{step^{LPC}}{\rightarrow} & \left(\mathcal{F}_e, \mathcal{S}^{LPC}_e\right) \\
\mathcal{R}^{LPC} &= \left\{(0,0), (0,+1), (0,-1), (+1,0), (-1,0)\right\} \\
\mathcal{S}^{LPC}_e &= e + \mathcal{R}^{LPC} \\
\mathcal{F}_e &: \mathbb{R}^5 \mapsto \mathbb{R} \\
&\quad (x_1, x_2, x_3, x_4, x_5) \mapsto \frac{\sum_{u=1}^{5} x_u}{5} \\
\mathcal{M}^{i+1}_{LPC}[e] &= \frac{1}{5}\Big(\mathcal{M}^{i}_{LPC}[e + (0,0)] + \mathcal{M}^{i}_{LPC}[e + (0,+1)] \\
&\quad + \mathcal{M}^{i}_{LPC}[e + (0,-1)] + \mathcal{M}^{i}_{LPC}[e + (+1,0)] \\
&\quad + \mathcal{M}^{i}_{LPC}[e + (-1,0)]\Big)
\end{aligned}
$$

Table 2.12: $\mathcal{HUA}$ model of the Laplace (LPC) stencil application that is described in pseudo-code in Figure 2.6

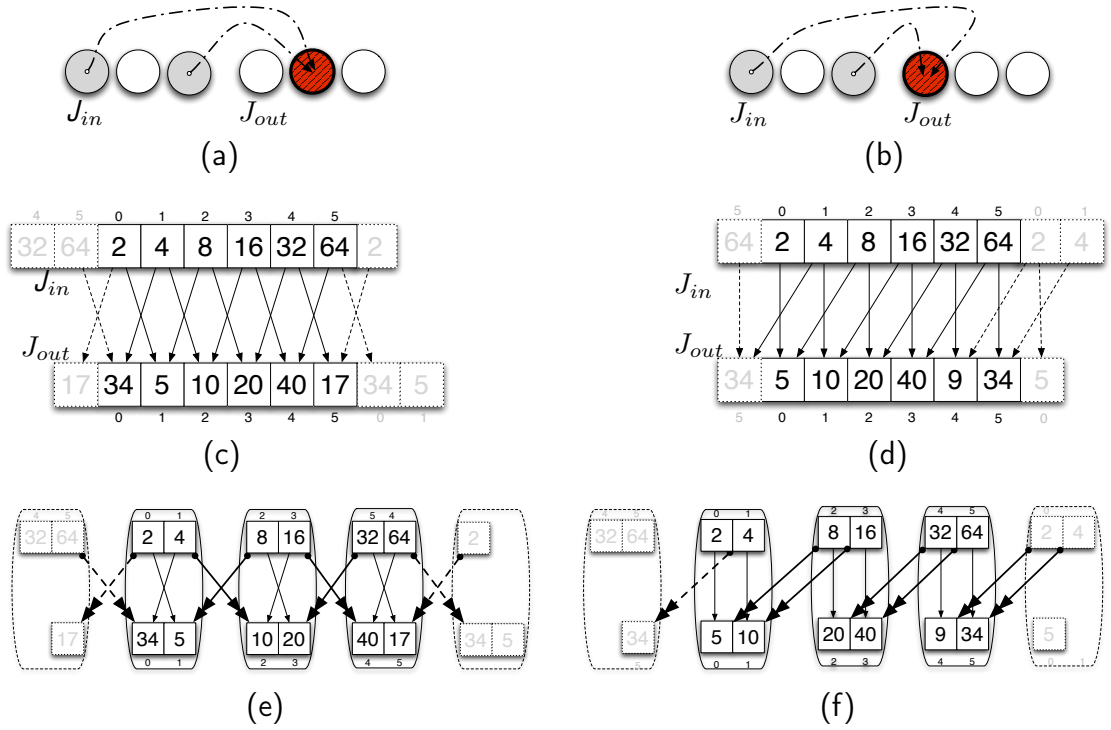


Figure 2.21: Representations, in one-dimensional space, of both the Jacobi stencil (Fig. 2.21(a)) and its modified version (Fig. 2.21(b)), where the application point has been shifted by one position. Example of one step computation of the original (Fig. 2.21(a)) and modified Jacobi (Fig. 2.21(b)) over a circular vector representing a toroidal domain. Finally a parallelization of the original (Fig. 2.21(e)) and modified (Fig. 2.21(f)) Jacobi to highlight the different communication patterns.

We used this example in the Introduction Chapter to present the concept of relaxed equivalence. Indeed, from the analysis of a single step computation that was performed on the same input by the two versions of the Jacobi (compare Figure 2.21(b) and Figure 2.21(d)), it turns out that the two results are equivalent except for a redistribution of the values in the data structures. More precisely we find that, if in the original stencil an output value is stored at position i, in the modified version the same value is sored at position $i-1$. We now focus on using the relative shape in order to prove that the stencil shapes of the two versions are linked by the same relation that exists between the resulting vectors.

Table 2.13 and Table 2.14 report the $\mathcal{HUA}$ model of the two versions of the Jacobi. The impact of the different application points is reflected in the two different relative shapes. The same effect was also represented in the general model, nevertheless the $\mathcal{HUA}$ captures in a better and more concise way, through the relative shape entity, the peculiar nature of the differences between the two shapes: the shapes are

equivalent except for a rigid translation. Indeed, exploiting $\mathcal{R}^{Jo}$ and $\mathcal{R}^{Jm}$, the two relative shapes, we can demonstrate that:

$$\mathcal{S}_i^{Jm} - 1 = \big(x + \{0, +2\}\big) - 1 = x + \{+1, -1\} = \mathcal{S}_i^{Jo}$$

$$\mathcal{S}_i^{Jo} + 1 = \big(x + \{+1, -1\}\big) + 1 = x + \{0, +2\} = \mathcal{S}_i^{Jm}$$

The rigid translation which links the two relative shapes is the same which links the two resulting vectors of Figure 2.21(c) and Figure 2.21(d). The presented proof can be considered as a preview of what we will present in the next Chapter.

We conclude by stressing that there are no expressiveness differences between the $\mathcal{HUA}$ and the general model presented in the previous Chapter. What we have previously proved could also be done by using the general model, but it would require a more complex formalism.

In order to show the potential of the relative shape, we purposely chose the original and modified Jacobi example. The aim was to focus the discussion, once again, on the concept of relative equivalence. Indeed we wish to formalize the concept in the $\mathcal{HUA}$ model as follows.

Definition 2.5.2 (Relaxed Computational Equivalence in the $\mathcal{HUA}$ Model). Let ψ and χ be two stencil-based computations which are expressed in the $\mathcal{HUA}$ model. Then let $\mathcal{M}_\psi$, the spatial structure of ψ, and $\mathcal{M}_\chi$, the spatial structure of χ, be equivalent. A step $step_i$ of ψ and a step $step_j$ of χ are defined as relaxed-equivalent if a linear transformation ϕ, featuring the identity as rotation matrix, exists such that:

$$\forall\, e \in \mathcal{M}_\psi \wedge e \in \mathcal{M}_\chi,\ \ \mathcal{M}_\psi^i[e] = \mathcal{M}_\chi^j[e] \Rightarrow \mathcal{M}_\psi^{i+1}[e] = \mathcal{M}_\chi^{j+1}[\phi(e)]$$

The two $\mathcal{HUA}$ stencils ψ and χ are **relaxed-equivalent** if all their steps are relaxed-equivalent. Finally, a stencil transformation which results in a stencil that is relaxed-equivalent to the original one is classified as **relaxed-safe**.

Informally, two steps, that feature the same index space, are relaxed-equivalent if, from the same input working domain, they return two output working domains that feature the same values but which are possibly stored in different positions.

From one output working domain it is possible to reconstruct the other, knowing the linear transformation that is associated to relaxed equivalence.

$1D$ *Original Jacobi (Jo)* $\mathcal{HUA}$ step model

$$
\begin{aligned}
\forall e = (x) \in \mathcal{M}_{Jo} \quad &\overset{step^{Jo}}{\rightarrow} \quad \left(\mathcal{F}_e, \mathcal{S}_e^{Jo}\right) \\
\mathcal{R}^{Jo} \quad &= \quad \big\{(+1), (-1)\big\} \\
\mathcal{S}_e^{Jo} \quad &= \quad e + \mathcal{R}^{Jo} \\
\mathcal{F}_e \quad &: \quad \mathbb{R}^5 \mapsto \mathbb{R} \\
&\qquad\quad (x_1, x_2) \mapsto \frac{\sum_{u=1}^{2} x_u}{2} \\
\mathcal{M}_{Jo}^{i+1}[e] \quad &= \quad \frac{1}{2}\Big(\mathcal{M}_{Jo}^{i}[e+1] + \mathcal{M}_{Jo}^{i}[e-1]\Big)
\end{aligned}
$$

Table 2.13: $\mathcal{HUA}$ model of the Laplace (LPC) stencil application that is described in pseudo-code in Figure 2.6

$1D$ *modified Jacobi (Jm)* $\mathcal{HUA}$ step model

$$
\begin{aligned}
\forall e = (x) \in \mathcal{M}_{Jm} \quad &\overset{step^{Jm}}{\rightarrow} \quad \left(\mathcal{F}_e, \mathcal{S}_e^{Jm}\right) \\
\mathcal{R}^{Jm} \quad &= \quad \big\{(0), (-2)\big\} \\
\mathcal{S}_e^{Jm} \quad &= \quad e + \mathcal{R}^{Jm} \\
\mathcal{F}_e \quad &: \quad \mathbb{R}^2 \mapsto \mathbb{R} \\
&\qquad\quad (x_1, x_2) \mapsto \frac{\sum_{u=1}^{2} x_u}{2} \\
\mathcal{M}_{Jm}^{i+1}[e] \quad &= \quad \frac{1}{2}\Big(\mathcal{M}_{Jm}^{i}[e] + \mathcal{M}_{Jm}^{i}[e-2]\Big)
\end{aligned}
$$

Table 2.14: $\mathcal{HUA}$ model of the Laplace (LPC) stencil application that is described in pseudo-code in Figure 2.6

Chapter 3

The Complete $\mathcal{HUA}$ Architecture

Abstract

In this Chapter, we analyze the complete architecture of a framework for programming $\mathcal{HUA}$ stencil-based applications. The software architecture is divided into four levels called *functional dependency*, *partition dependency*,*concurrent level* and *firmware level*. At the *functional dependency level*, a stencil-based application is represented in terms of the structured model, where the functional dependencies are highlighted. At the *partition dependency level*, the $\mathcal{HUA}$ stencil description is translated in terms of space partitions. At the *concurrent level*, a representation of the stencil in terms of communicating processes is extracted. Finally, at the *firmware level*, the program is compiled for a specific architecture.

The Chapter is structured as follows. Section 3.3 presents our reference architecture while Section 3.2 and Section 3.3 analyse in detail the partition dependency and concurrent levels.

Finally, Section 3.4 analyzes a particular stencil application at each of the architecture levels. The Section introduces the best methods in the literature for optimizing the number of communications in the implementation of a $\mathcal{HUA}$ stencil.

Contents

3.1 The Reference Architecture

In the previous Chapter, we presented the stencil structured model for a certain class of stencil-based applications. The model provides some mechanisms to programmers in order to describe their applications. Those mechanisms alone are not sufficient to produce a program which can be run on a target parallel architecture.

A stencil representation, in terms of the structured model, features a high degree of abstraction with respect to an executable program described at firmware level. The abstraction gap therefore has to be filled with a process of concretization, where each mechanism of high abstraction is implemented with mechanisms featuring a lower level of abstraction.

We rely on a concretization method based on a hierarchical approach. Indeed, we consider an architecture divided into different levels of abstractions. Each level features its own set of mechanisms and its own language to manage the mechanisms. The concretization strategy consists in the transformation of a program described at a certain level into an equivalent one described at the level immediately below.

By iterating the concretization strategy level by level, the high level description of the stencil-based application, from which we can easily study useful properties for optimization purposes, is transformed into an equivalent executable program expressed at the firmware level.

Figure 3.1 shows our four level reference architecture. The process of concretization for a generic stencil application can be schematized as follows:

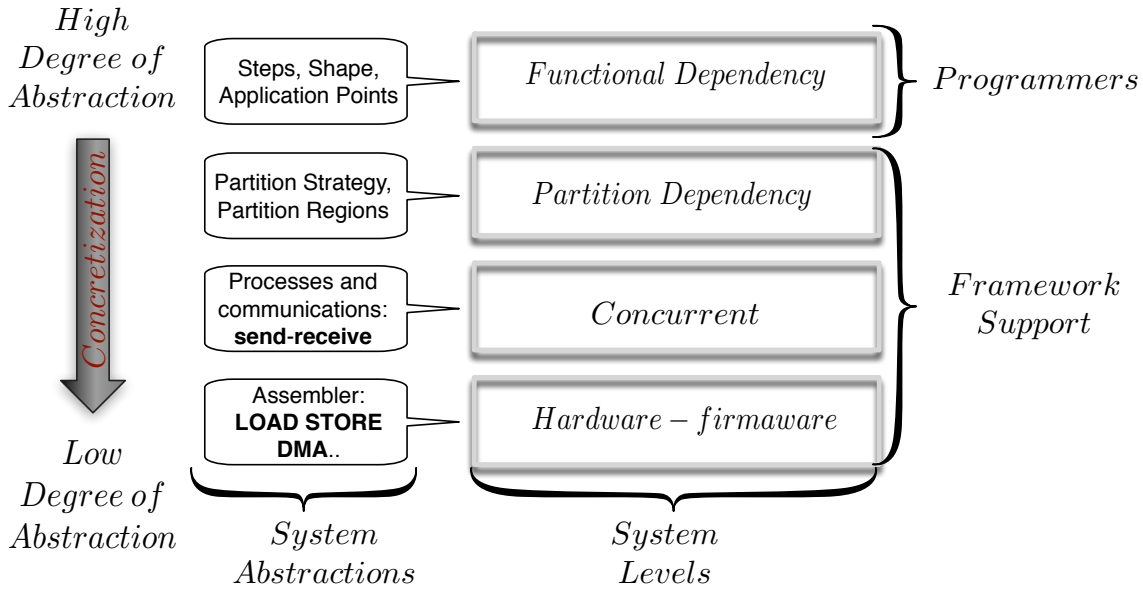


Figure 3.1: Representation of reference architecture for implementing $\mathcal{HUA}$ stencils.

I At the top of our architecture, we have the **functional dependency level**,

which features a language which, thanks to some nice syntax, allows programmers to define a stencil application according to the structured stencil model.

At this level, the description of a stencil program is mainly focused on the functional dependencies between spatial elements, described in terms of shapes and application points, hence the name of the level.

The mechanisms featured by the functional dependency level are the ones that we described in the previous Chapter when presenting the structured stencil model: steps, working domains, shapes, application points, etc.

II Once a description of the application has been established according to the structured stencil model, it is translated into the **partition dependency level**. At this level, a partition strategy for the elements of the spatial structure is selected and all the dependencies between spatial elements are translated into dependencies between partitions.

III At the **concurrent level**, a process is associated to each partition. Exploiting a message passing formalism, the partition dependencies are resolved with appropriate communications between processes. Furthermore, the data structures needed to represent the working domain are selected.

IV Finally the program described in a message passing language can be directly compiled at **firmware level** in order to obtain an executable program for a target parallel architecture.

The representation of the architecture in Figure 3.1 highlights where the programmers of a stencil-based application are involved. Once they have produced a stencil description in the structured stencil model, they do not have to manage any aspects of the concretization phase. A set of tools, that we call framework support, can be defined to automatically produce the executable program of the application.

With respect to the presented architecture, our studies focus on analysing the impacts produced by properties at the functional dependency level at the concurrent level. In particular, the scope behind the following Sections is to understand how the functional dependencies between elements influence the number of communications between processes. Moreover, the next Chapter extends the result in order to define how new transformations at the dependency level can reduce communication overheads.

3.2 The World of Partitions

A delicate phase in data parallel application development is partitioning. Important performance aspects of a parallel program, like the degree of parallelism, the number of communications and also their sizes, depend on the specific partitioning strategy adopted. This is true both for distributed architecture and shared memory, where a communication has to be considered as read or write operations.

For general data parallel applications, the problem of defining a partitioning that maximises the performance is an NP problem [25, 26].

In the case of $\mathcal{HUA}$ space invariant stencils, where all the computations are focused on one single spatial structure and functional dependencies have a well defined geometric structure, we confine our attention to regular partitioning.

In Section 3.2.1 we formally define some working hypotheses on partitioning strategies, and in Section 3.2.2 we present some conventions and notations that we exploit to manage partitions.

Finally in Section 3.2.3 we classify the elements of a partition into regions according to target stencil. Moreover we formalize some mechanisms for representing partition dependencies.

3.2.1 Working Hypotheses on Partitioning Strategies

In our study, we consider two hypotheses on partitioning. The first concerns a restriction on the types of regular partitioning on which we focus. The second introduces some limitations on the features of the resulting partitions, with respect to the target stencil. We start by considering the restrictions on partitioning strategies.

Working Hypothesis 3.1. In our study we confine our attention to **regular** and **block** partitioning strategies with the following bounds.

I The partitioning concerns all space dimensions.

II All the resulting partitions feature the same space volume.

It is well known that it is sometimes preferable, for performance reasons, not to consider all the dimensions of a working space for partitioning [54]. Taking again the case of the two-dimensional Laplace stencil, the previous sentence claims that there are cases in which it can be convenient to split the matrix by rows instead of by blocks. In other words, better performance can be achieved when partitioning of rows is avoided. This kind of strategy is not taken into account by Working Hypothesis 3.1.

It is worth mentioning that whether the first approach is convenient for the second or vice versa depends on different parameters like the mean computation latency associated with the update of a domain element value and the degree of parallelization which we are targeting. A simple model for this problem, in the case

of a two-dimensional working domain, is presented by Wilkinson and Allen [54]. The result is that for high degrees of parallelization, it is preferable to partition over all the dimensions.

In our study, we keep focusing only on this last case, because it is the most complicated for communication pattern and, consequently, it is more interesting as a study case. Moreover our results can be directly reused to implicitly define the computation in a space with a lower number of dimensions. For instance, it is possible to model the two-dimensional Jacobi over a one-dimensional space where the type of spatial element is equivalent to a row of the original problem.

Along with the previous working hypothesis about regular partitioning, we take into consideration another bound about the volume of a partition.

Working Hypothesis 3.2. Consider an application described by a $\mathcal{HUA}$ stencil ψ. In our study, we consider only those partition strategies resulting in partitions whose volume is larger than the volume of the relative shape $\mathcal{R}$ associated with ψ.

In the example of the Laplace-based application, the previous working hypothesis imposes the condition that the generic partition provides a surface wider than the four by four surface which circumscribes the relative cross shape.

The restriction is realistic, in the sense that the introduced bound is respected by real stencil-based applications for two main reasons.

I Working domains are usually extremely wide relative to the volume of the relative shape of a stencil. Let us consider a partitioning which results in partitions whose volume is equal to the volume of the stencil relative shape. In such a configuration, the number of resulting partitions would be extremely high.

II Foremost, because the computation for a single element features in most cases a low latency, with too small partitions there would be no chance of overlap between communications and computations; this would result in low efficiency.

3.2.2 Conventions and Notations for Partitions

In the analysis of the impact of communication on stencil-based applications, partitions play a fundamental role. To discuss the spatial properties of a partition, we introduce in this Section some useful notations and conventions.

Considering the previous Working Hypothesis 3.1, we can give a formal definition of the partition space:

Definition 3.2.1 (Partition Space). Let ψ be a space invariant $\mathcal{HUA}$ stencil defined over an n-dimensional space and let $\mathcal{W}^{\psi}$ be its working domain. The partition space $\mathcal{P}$ of $\mathcal{W}^{\psi}$ is a subset of $\mathbb{N}^n$ where each point represents one working domain partition:

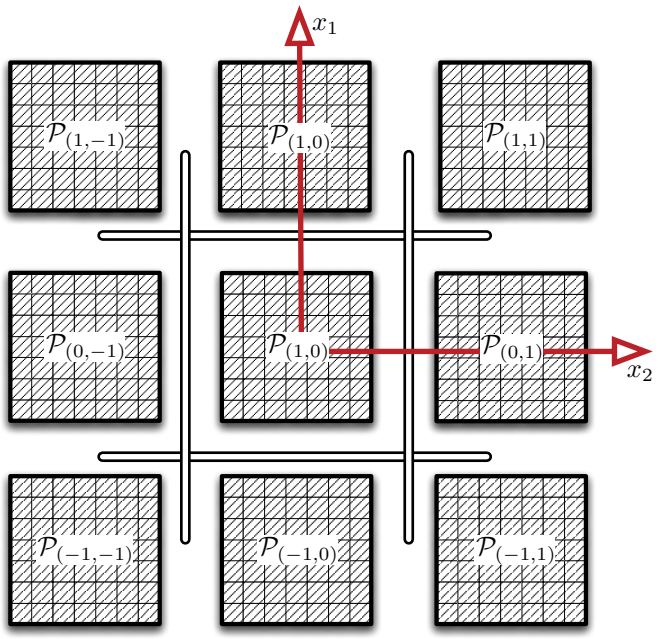


Figure 3.2: Representation of the partition reference system for a two-dimensional working domain.

$$\mathcal{P} = \overbrace{\frac{\mathbb{Z}}{p_1} \times \frac{\mathbb{Z}}{p_2} \times \ldots \frac{\mathbb{Z}}{p_n}}^{n\ factors}$$

The vector $p = (p_1, \ldots, p_n)$ is the **partition space length** vector. The i-th component of p defines how many partitions have been considered along the i-th space dimension. The generic element $\mathcal{P}_\alpha \in \mathcal{P}$ represents a subset of the spatial structure according to its position in $\mathcal{P}$.

The partitioning and functional dependencies between elements defined by a stencil produces what we call **partition dependencies**. The effects of these dependencies is a mandatory information exchange between partitions in order to update all the elements during a single step.

With a fixed partition $\mathcal{P}_\alpha$, the set of other partitions that store necessary information in order to update all the element of a $\mathcal{P}_\alpha$ depends both on the partitioning strategy and on the stencil shape being considered. By considering Working Hypothesis 3.1 and Working Hypothesis 3.2, we derive the following important property.

Property 3.2.1 (Neighbour Interactions). *A generic partition $\mathcal{P}_\alpha$ can feature partition dependencies only with its spatial neighbour partitions.*

We therefore need some mechanisms to identify in partition space the neighbours of a partition $\mathcal{P}_\alpha$. For the sake of simplicity, we adopt a relative reference system

centred on $\mathcal{P}_\alpha$, in this system partitions are represented by vectors centred on $\mathcal{P}_\alpha$. Figure 3.2 shows the partition reference system for a two-dimensional working domain.

We can formally define the neighbour set of a partition as follows.

Definition 3.2.2 (**The Neighbour Set).** Let $\mathcal{P}$ be a partition space and let $\mathcal{P}_\alpha$ be a generic partition in the space. Let us consider now a relative reference system centred on $\mathcal{P}_\alpha$. In the new system, we define the **neighbour index set** of $\mathcal{P}_\alpha$ as the following vector set:

$$NI(\mathcal{P}_\alpha) = \{\forall(\beta_1, \ldots, \beta_n) \mid \forall i \ \beta_i \in \{-1, 0, 1\} \ and \ \exists j \ |\beta_j \neq 0\}$$

We call the generic element of the neighbour index set the **movement vector**. Associating with each index the corresponding neighbour partition, we define the **neighbour set** as follows:

$$NS(\mathcal{P}_\alpha) = \{\forall \mathcal{P}_{((\beta_1,\ldots,\beta_n))} \mid (\beta_1, \ldots, \beta_n) \in NI(\mathcal{P}_\alpha)\}$$

The neighbour index set groups all the space vectors that reference the neighbours of the origin of $\mathbb{Z}$, the set of integer numbers. The elements of the set are then used in the neighbour set to identify the neighbour partitions in the relative system centred on the generic partition $\mathcal{P}_\alpha$.

For an n-dimensional working domain, we can claim that the cardinality of $NS(\mathcal{P}_\alpha)$ is equal to $3^n - 1$. Indeed, the neighbour index set, which has the same cardinality as the neighbour set, is composed of all combinations of strings of length n which can be composed by exploiting the alphabet $\{-1, 0, 1\}$, apart from the string composed of all zeros. The zero string is associated with $\mathcal{P}_\alpha$ which does not belong to $NS(\mathcal{P}_\alpha)$.

3.2.3 Regions: the Classification of Partition Elements

We know that, when a partition strategy is selected, the functional dependencies between spatial elements are translated into partition dependencies. In this section we focus on the analysis of this transformation and define a classification of the partition elements into distinct regions.

A first subdivision of a partition can be computed with respect to the functional dependencies of its local elements. We therefore consider the logical splitting of partition elements into two regions called *incoming independent region* and *incoming dependent region.* The first is defined as follows.

Definition 3.2.3 (**Incoming Independent Region).** Let $\mathcal{P}_\alpha$ be a generic partition of a working domain and let ψ be a generic $\mathcal{HUA}$ stencil. We define the *incoming independent region (IIR)* of $\mathcal{P}_\alpha$ associated to ψ as the following set:

$$IIR(\mathcal{P}_\alpha) = \left\{\forall e^{local} \in \mathcal{P}_\alpha \mid e^{remote} \in S^\psi(e^{local}) \Rightarrow e^{remote} \in \mathcal{P}_\alpha\right\}$$

Less formally, the region includes all those elements whose associated shape is completely composed of elements of the partition. Therefore the elements of the incoming independent region do not require any other information from the outside world (represented as the elements of all other partitions), hence the name we selected for the region.

Figure 3.3(a) reports the representation of a stencil that we use to graphically show the regions of a partition. On purpose, we selected such an uncommon stencil, whose shape does not feature any symmetry with respect to the application point, in order to avoid possible erroneous conclusions.

Figure 3.3(a) highlights in white the elements of a general partition which, according to the unusual stencil being considered, belong to the incoming independent region. What we can see from the Figure is that the incoming independent region is a convex set. It is easy to prove that this observation can be extended to all $\mathcal{HUA}$ stencils.

The second part into which we logically divide a generic partition is characterized as follows:

Definition 3.2.4 (**Incoming Dependent Region).** Let $\mathcal{P}_\alpha$ be a generic partition of a working domain and let ψ be a generic $\mathcal{HUA}$ stencil. We define the *incoming dependent region (IDR)* of $\mathcal{P}_\alpha$ associated to ψ as the following set:

$$IDR(\mathcal{P}_\alpha) = \left\{ \forall e^{local} \in \mathcal{P}_\alpha \mid \exists e^{remote} \in S^\psi(e^{local}) \text{ and } e^{remote} \notin \mathcal{P}_\alpha \right\}$$

This region, which is complementary to the incoming dependent region relative to the partition space, collects elements that, in order to be updated according to the stencil shape, require values of some remote elements, i.e. elements that are located in other partitions. Figure 3.3(b) highlights the dependent incoming region elements in gray.

As is evident from the Figure and in contrast to the incoming independent region, the new region is not a compact set. Instead, the region features the property of being concentrated next to the partition borders, or in other words that at least one of the borders of the region is also the border of the partition.

We formally model and prove the previous property with the following Theorem:

Theorem 3.2.1 (**Incoming Dependent Region Placement).** *Let the element e^{local} belong to the incoming dependent region of a partition $\mathcal{P}_\alpha$ with respect to a stencil ψ. Let δ be one of the partition borders that the ψ shape intersects when it is applied to e^{local}. All partition points between e^{local} and δ belong to the incoming dependent region.*

Proof. First of all, the existence of δ is ensured by the fact that the element e^{local} belongs to the incoming dependent region. In fact, by the definition of an incoming dependent region, when one of its element is considered as an application point, the associated shape features at least one element which does not belong to the partition.

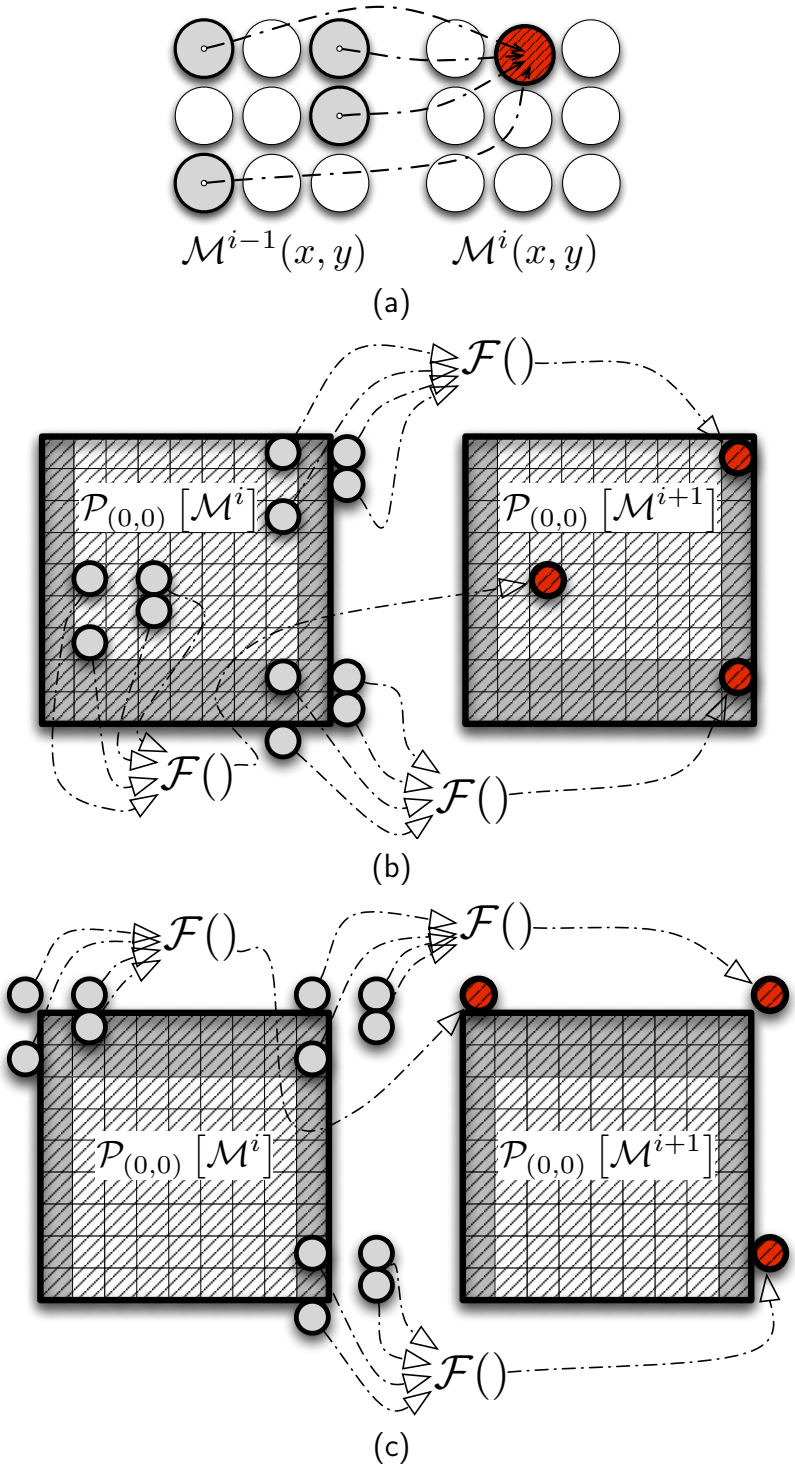


Figure 3.3: Graphical representation of stencil shape (3.3(a)), incoming dependent (colored gray) and independent regions (3.3(b)), and finally outgoing dependent (colored gray) and independent regions (3.3(c))

We can conclude that the shape has to intersect at least one of the partition borders. We have therefore demonstrated the correctness of the theorem statement, we now proceed to proving its the validity.

The key point of the proof is given by the uniformity property of $\mathcal{HUA}$ stencils. Consider e^{remote} one of the elements that belongs to the shape applied to e^{local} but does not belong to the partition $\mathcal{P}_\alpha$; in other words, e^{local} has a functional dependency on e^{remote}. We can assert that all the elements of $\mathcal{P}_\alpha$ between e^{local} and e^{remote} belong to the incoming dependent region. Suppose e_m is one of these elements. If the shape of the stencil applied to e^{local} features elements out of the partition bound, the same shape applied to e_m, which is closer to the partition border by construction, cannot be completely inside the partition. If the shape centered on e_m also has elements that do not belong to its partition, e_m is classified as an element of the incoming dependent region. □

It is worth mentioning that another key piece of information comes from the incoming dependent region. The elements of the region define a set, let us call it the **incoming set**, composed of non local elements. The set is the union of the remote elements of all the shapes whose application point belongs to the incoming dependent region.

The incoming set models the non local information that a generic partition has to acquire in order to update all its elements. The way the set is scattered between the neighbouring partitions defines what we call the **set of incoming partition dependencies**. A formal definition follows.

Definition 3.2.5 (**The Incoming Partition Dependency Set).** Let ψ be a $\mathcal{HUA}$ stencil and let $NS(\mathcal{P}_\alpha)$ be the set of neighbours of a generic partition $\mathcal{P}_\alpha$. We define the incoming partition set Δ^{in} as the following subset of $NS(\mathcal{P}_\alpha)$:

$$\Delta^{in} = \left\{\forall \mathcal{P}_\beta \in NS(\mathcal{P}_\alpha) | \exists e^{local} \in \mathcal{P}_\alpha, e^{remote} \in \mathcal{P}_\beta and\ e^{remote} \in \mathcal{S}^\psi(e^{local})\right\}$$

In a manner that is orthogonal to the two previous regions, which have been characterized according to the dependencies on the elements of other partitions, we classify the partition into two other regions: *outgoing dependent* and *outgoing independent*.

Definition 3.2.6 (**Outgoing Dependent Region).** Let $\mathcal{P}_\alpha$ be a generic partition of a working domain and let ψ be a generic $\mathcal{HUA}$ stencil. We define the *outgoing dependent region (ODR)* of $\mathcal{P}_\alpha$ associated to ψ as the following set:

$$ODR(\mathcal{P}_\alpha) = \left\{\forall e^{local} \in \mathcal{P}_\alpha \mid \exists e^{remote} \in S^\psi(e^{local})\ and\ e^{remote} \notin \mathcal{P}_\alpha\right\}$$

The region that is highlighted in Figure 3.3(c) in gray groups the elements required by other partitions to finish updating all their elements. The value of each element of this region has to be provided to one or more neighbouring partitions. The second region is characterized as follows.

Definition 3.2.7 (**Outgoing Independent Region).** Let $\mathcal{P}_\alpha$ be a generic partition of a working domain and let ψ be a generic $\mathcal{HUA}$ stencil. We define the *outgoing independent region (OIR)* of $\mathcal{P}_\alpha$ associated to ψ as the following set:

$$OIR(\mathcal{P}_\alpha) = \left\{\forall e^{local} \in \mathcal{P}_\alpha \mid \forall e^{remote} \notin \mathcal{P}_\alpha \Rightarrow e^{local} \notin S^{\psi}(e^{remote})\right\}$$

This region is complementary to the outgoing dependent region relative to the partition space. It collects all the elements whose values are not going to be exchanged with any other partition. As the incoming independent region, this region is a compact set. The region is highlighted in white in Figure 3.3(c).

A key piece of information is extracted from the outgoing dependent region. The way the elements of the region are requested by neighbouring partitions defines what we call the **set of outgoing partition dependencies**. A formal definition follows.

Definition 3.2.8 (**The Outgoing Partition Dependency Set).** Let ψ be a $\mathcal{HUA}$ stencil and let $NS(\mathcal{P}_\alpha)$ be the set of neighbours of a generic partition. We define Δ^{out}, the outgoing partition set, as follows:

$$\Delta^{out} = \left\{\forall \mathcal{P}_\beta \in NS(\mathcal{P}_\alpha) | \exists e^{remote} \in \mathcal{P}_\beta, e^{local} \in \mathcal{P}_\alpha, e^{local} \in \mathcal{S}^{\psi}(e^{remote})\right\}$$

From a comparison of Figure 3.3(b) and Figure 3.3(c), it is evident that the incoming dependent region and the outgoing dependent region are composed of different partition elements. This observation can be symmetrically reported for the two other remaining regions.

We will see in the next Section that for $\mathcal{HUA}$ stencils whose shapes feature a central symmetry with respect to the application point, the incoming and outgoing dependent regions coincide.

3.3 The World of Concurrency

A stencil-based application at the concurrent level is defined by a set of communicating processes. The partition dependencies defined by Δ^{in} Δ^{out}, the incoming and outgoing partition dependency sets, are transformed or better resolved by communications between processes.

In this section we first describe $\mathcal{LC}$, which is the language we target as a concurrent language, then we present a cost model associated with the language which allows us to estimate the performance of a program especially in terms of communication impacts. Finally, we present some conventions and notations that we use when working at the concurrent level.

3.3.1 The $\mathcal{LC}$ language

$\mathcal{LC}$ is the language we define for the concurrent level; it is based on a message passing paradigm and describes abstract processes communicating according to a local environment model. $\mathcal{LC}$ was born as a tailored minimal formalism of the *CSP* of Hoare [22].

An $\mathcal{LC}$ parallel program statically declares a set of processes, which can also be described parametrically with respect to some identifiers. An in-depth description of the $\mathcal{LC}$ language and a presentation of an implementation for the Cell multi-core architecture are the subjects of Appendix A.

An $\mathcal{LC}$ channel is unidirectional and typed: it represents a queue where a set of producer processes insert messages and a single consumer process extracts data. The type of channel and the type of messages in its queue must match. Channels can implement either symmetric communications, when only one producer is defined, or asymmetric communications.

An important characteristic of $\mathcal{LC}$ channels is the degree of asyncronicity, which is a static integer parameter defining the maximum number of non blocking send operations that can be performed when no receive operation is invoked. When the parameter is equal to zero, the channel is by definition synchronous. The degree of asyncronicity is statically defined and has to be guaranteed by the implementation of the language for the lifetime of the $\mathcal{LC}$ program.

In order to target high performance computing, an important aspect is the possibility of featuring communication mechanisms that can overlap communications and computations. This depends on the characteristics of the target architecture and on the implementation of the communication mechanisms. As proved by the experiments given in Chapter A, our $\mathcal{LC}$ implementation on the Cell multi-core architecture, called $\mathcal{M}ammu\mathcal{T}$, guarantees that the best overlapping provided by the firmware architecture can be exploited.

Finally, $\mathcal{LC}$ includes an alternative guarded command, similar to *CSP*, to manage non deterministic aspects; this is a key relevant construct for dynamic load balancing aspects.

3.3.2 Communication Cost Model

We consider a cost model associated to the concurrent level. The aim of the model is to wrap the performance feature of a target firmware architecture into a small number of parameters. This approach allows an easy analytic analysis of the performance associated with an $\mathcal{LC}$ parallel program without requiring knowledge of the specific firmware architecture.

We consider a simple and widely used cost model for communications. We model the communication latency of a send operation as a constant setup time plus a transfer time that is proportional to the length of the message. To a single send operation, we associate the following parameter:

$$T_{send}(msg) = t^{snd}_{setup} + t_{transm} * s_{msg}$$

msg is the message and s_{msg} its size. Moreover t^{snd}_{setup}, t_{transm} are two constants that depend on the specific physical architecture. In our analysis, we consider the two constants as unknown variables of the system. In other words all our considerations on communication impacts will be parametric with respect to both t^{snd}_{setup} and t_{transm}.

The model has to also consider the overlapping aspects of communication and computation. Indeed, for certain implementations, a part of the whole communication latency could partially overlap the computation. For the sake of simplicity we suppose that in the defined $T_{send}(msg)$, the component $t_{transm} * s_{msg}$ exactly corresponds to the latency that can completely overlap some computations.

Finally, concerning receive operations, we model their latency as a constant that is independent of message size:

$$T_{recv}(msg) = t^{recv}_{setup}$$

In the model that we are targeting, we concentrate all the performance variables related to the size of the massage only in the send phase. We follow this strategy for two main reasons; firstly we can manage a more simple model and secondly this approach is well suited for those implementations of a message passing communication languages which guarantee zero-copy communication protocols.

We report an example in Appendix A, where we present an implementation of a communication library for the Cell B.E. In this case, as the analyses of the exploited protocols and the presented cost model prove, we have reduced the receive latency to a simple constant which is independent of the message size.

In Appendix A, a discussion about the relation between the expressiveness of communication languages and an implementation featuring zero-copy protocols is also reported.

Before proceeding, we consider worthwhile to clarify and underline that the choice of a cost model which includes all the dependencies of the massage size in the send performance is not going to influence the results we are going to introduce.

3.3.3 $\mathcal{LC}$ and $\mathcal{HUA}$ Stencils

When implementing a $\mathcal{HUA}$ stencil-based application, there are three phases that have to be considered.

I Data are distributed in such a way that each process acquires its own data partition.

II Partition dependencies are resolved through inter process communications and each process updates its element values.

III Final values are gathered to rebuild the result data structures.

The focus of our studies is on the second of the three phases. We are interested in understanding the possible ways in which we can transform, as efficiently as possible, the mechanism at the functional dependency level into mechanisms at the concurrent level.

For the sake of simplicity, we refer to processes according to the name of their partition. Therefore with $\mathcal{P}_\alpha$ we identify both the partition and the process managing it.

Granted that our studies at the concurrent level mainly focus on the definition and analysis of the **incoming communication set** Δ^{in}_{com} and **outgoing communication set** Δ^{out}_{com}. The two sets define for a generic process $\mathcal{P}_\alpha$ those neighbouring processes interested in respectively incoming and outgoing communications in order to resolve the partition dependencies. Because partitions and processes share the same name, we can claim the following relations:

$$\Delta^{in}_{com} \subseteq \Delta^{in}$$
$$\Delta^{out}_{com} \subseteq \Delta^{out}$$

3.4 A Nine Point Stencil at Work

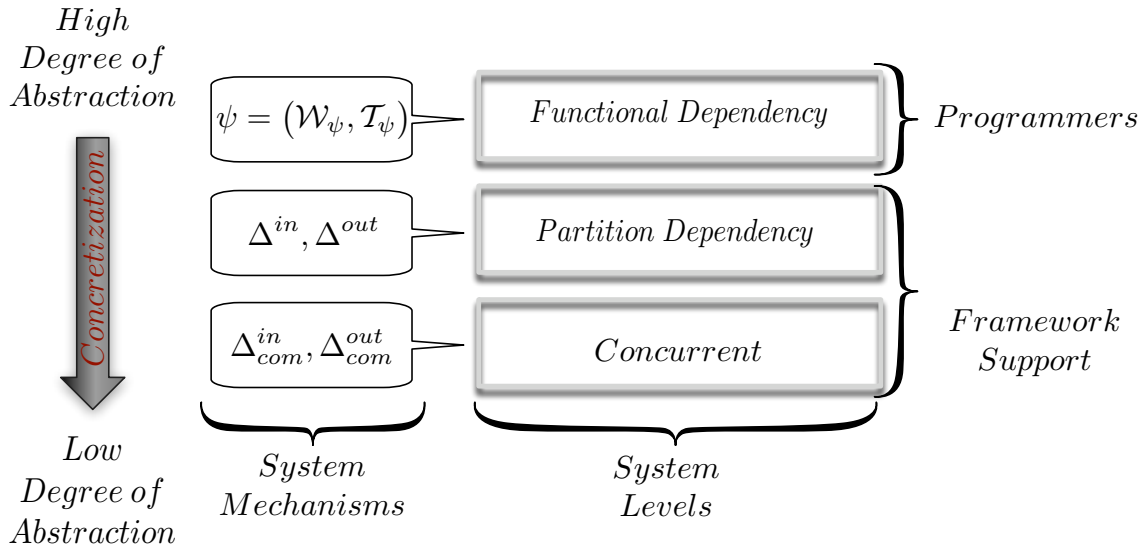


Figure 3.4: A representation of our reference architecture for implementing $\mathcal{HUA}$ stencils which highlights for each level the main mechanisms

In this Section we concentrate on the analysis of a particular stencil example defined over a two-dimensional space which is usually called the nine point stencil in the literature (for the sake of simplicity we call it *Nine*). This stencil is useful when studying the impacts of communication, because it can be considered to be a worst case test. A representation of a sequential application based on the nine point stencil is reported in pseudo-code in Figure 3.5.

In the following, we will focus the discussion on the analysis of the nine point stencil application, from the functional level to the concurrent one, as shown by Figure 3.4. The Figure also highlights for each architectural level the mechanisms we are going to define and study.

3.4.1 Functional Dependency Level

At the functional dependency level, the application is described by programmers in the $\mathcal{HUA}$ model, and therefore the dependencies between spatial elements are defined. The $\mathcal{HUA}$ description of the nine point stencil application is straightforward. Figure 3.1 reports the model of the step component, while Figure 3.6(a) shows a graphical representation of the stencil shape geometry.

```
double J_in[1024][1024], J_out[1024][1024];                                  1
load_working_domain_values(J_in);                                            2
for(i_step = 0; i_step < 5; i_step++){                                       3
                                                                             4
  forall((x, y) ∈ J_in){                                                     5
      J_out[x, y] = (J_in[x, y + 1] + J_in[x, y − 1]+                        6
            +J_in[x + 1, y] + J_in[x + 1, y + 1] + J_in[x + 1, y − 1]        7
            +J_in[x − 1, y] + J_in[x − 1, y + 1] + J_in[x − 1, y − 1])/8;    8
  }                                                                          9
                                                                             10
  swap(J_in, J_out);                                                         11
}                                                                            12
return_working_domain(J_out);                                                13
```

Figure 3.5: Representation in pseudo-code of a nine point stencil (Nine)-based application in a two-dimensional toroidal space. To keep the notation light, we assume the indices are automatically mapped onto the toroidal space, i.e. $J_{out}[-1,+1]$ is transformed into $J_{out}[+1023,+1]$

3.4.2 Partition Dependency Level

At the partition dependency level, a $\mathcal{HUA}$ stencil description in terms of functional dependencies between elements is translated into a description where the main objects are the partitions. Without any intervention from the programmer, functional dependencies are automatically re-described in terms of partitions, regions and incoming and outgoing partition dependency sets.

Figure 3.6(b) highlights, in a generic partition of the nine point stencil, the incoming independent and dependent regions; the first region is shown in white while the second is shown in gray. On the other hand, Figure 3.6(c) highlights the outgoing dependent and independent regions, respectively in gray and white.

In the case of the nine point stencil, the incoming and outgoing dependent regions are concentrated next to all the partition boundaries. Moreover, because the stencil shape features a central symmetry with respect to the application point, the two regions overlap perfectly, i.e. they are composed of the same partition elements.

Finally, it is worth mentioning that all the neighbouring partitions are included into both the incoming and outgoing partition dependency sets:

$$\Delta^{in} = \Delta^{out} = NS(\mathcal{P}_\alpha)$$

Therefore, for the nine point stencil, a generic partition has to both acquire and provide some form of information to all its neighbours. This is the feature that makes the nine point stencil a useful worst case example in studies of communication impacts.

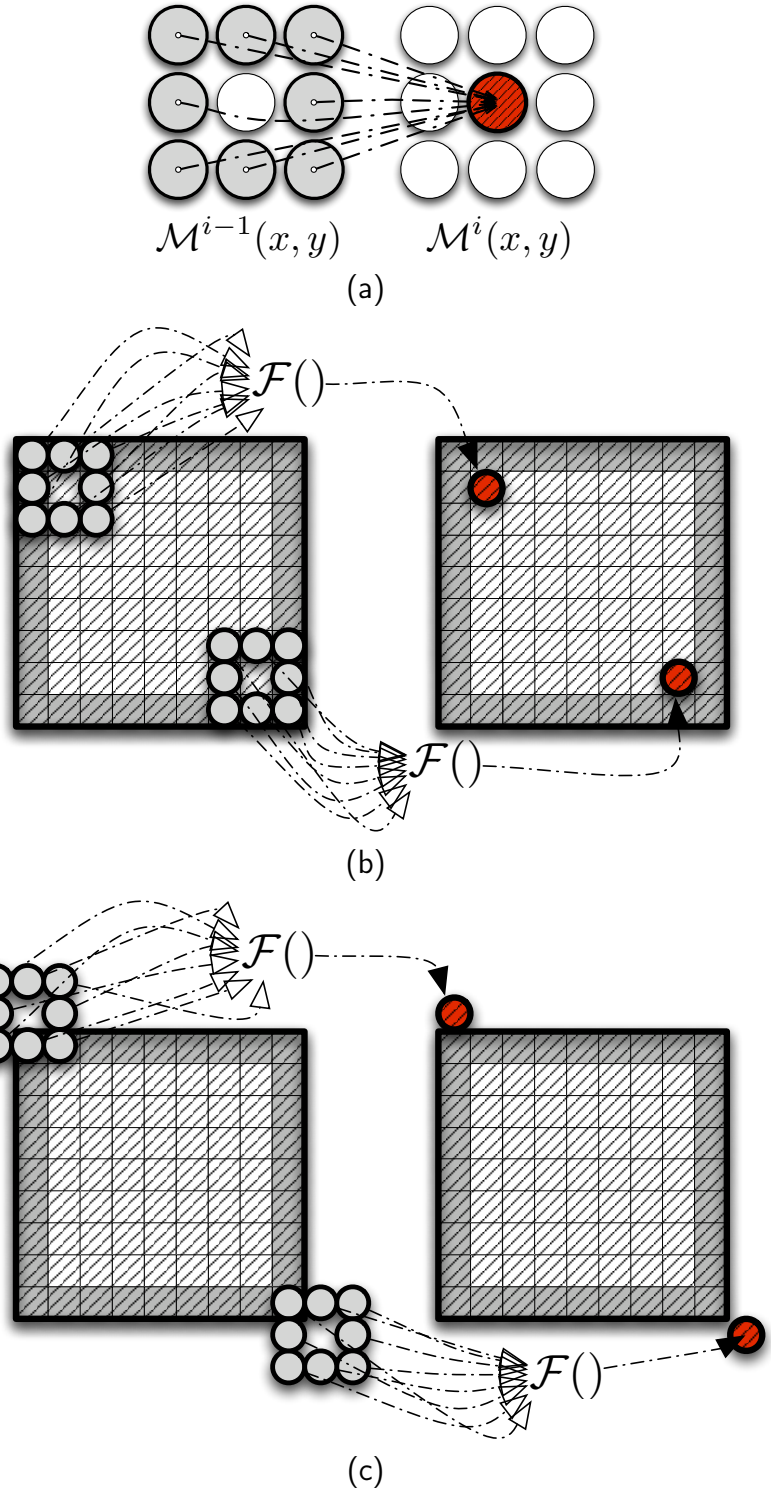


Figure 3.6: Graphical representation of stencil shape (3.6(a)), incoming dependent (colored in gray) and independent regions (3.6(b)), and finally outgoing dependent (colored in gray) and independent regions (3.6(c)) of a nine point stencil.

$\mathcal{HUA}$ Nine Point Stencil STEP(nine)

$$
\begin{array}{rcl}
\forall e = (x,y) \in \mathcal{M}_{nine} \overset{step_i^{nine}}{\rightarrow} & & \left(\mathcal{F}, \mathcal{S}_{ie}^{nine}\right) \\
\mathcal{R}^{nine} & = & \Big\{(x,y+1),(x,y-1), \\
 & & (x+1,y-1),(x+1,y),(x+1,y+1), \\
 & & (x-1,y-1),(x-1,y),(x-1,y+1)\Big\} \\
\mathcal{S}_{(x,y)}^{nine} & = & (x,y) + \mathcal{R}^{nine} \\
\mathcal{F} & : & \mathbb{R}^4 \mapsto \mathbb{R} \\
\mathcal{F} & : & \left(r_1, r_2, r_3, r_4\right) \mapsto \frac{\sum_{i=1}^{8} r_i}{8} \\
\mathcal{M}_{nine}^{i+1}[e] & = & \mathcal{F}\big(\mathcal{M}_{nine}^{i}[(x,y+1)], \mathcal{M}_{nine}^{i}[(x,y-1)], \\
 & & \mathcal{M}_{nine}^{i}[(x+1,y+1)], \mathcal{M}_{nine}^{i}[(x+1,y)], \\
 & & \mathcal{M}_{nine}^{i}[(x,y-1)], \mathcal{M}_{nine}^{i}[(x-1,y+1)], \\
 & & \mathcal{M}_{nine}^{i}[(x-1,y)], \mathcal{M}_{nine}^{i}[(x-1,y-1)], \\
 & & \mathcal{M}_{nine}^{i}[(x+1,y)], \mathcal{M}_{nine}^{i}[(x-1,y)]\big)
\end{array}
$$

Table 3.1: Step component in the structured stencil model of the nine point stencil application that is described in pseudo-code in Figure 3.5.

3.4.3 Concurrent Level: The *naive* Method

Passing from partition dependency level to concurrent level, the dependencies featured by partitions are resolved by inter process communication, for example through message passing mechanisms.

There are two ways to implement this information exchange. First, we consider the easiest which we call the *naive* method. This method resolves incoming and outgoing partition dependencies which manage its own partition of the spatial structure and has to both send and receive data to and from all its neighbour processes.

We report in Figure 3.7 the pseudo-code representing the behaviour of the generic process. For the sake of simplicity, in communication operations we just indicate the sender or receiver process, while we leave to Figure 3.8(a) and Figure 3.8(b) the burden of graphically highlighting the exact elements involved in the communications.

Each step in the *naive* method is made up of the following phases:

```
double J_in[512][512], J_out[512][512];                                          1
load_partition_values(J_in);                                                     2
for(i_step = 0; i_step < 5; i_step++){                                           3
                                                                                 4
 SEND(P_(-1,-1)); SEND(P_(-1,0)); SEND(P_(-1,1)); SEND(P_(0,-1));                5
 SEND(P_(0,1)); SEND(P_(+1,-1)); SEND(P_(+1,0)); SEND(P_(+1,+1));                6
                                                                                 7
 COMPUTE(Incoming independent region);                                           8
                                                                                 9
 RECV(P_(-1,-1)); RECV(P_(-1,0)); RECV(P_(-1,1)); RECV(P_(0,-1));                10
 RECV(P_(0,1)); RECV(P_(+1,-1)); RECV(P_(+1,0)); RECV(P_(+1,+1));                11
                                                                                 12
 COMPUTE(Incoming dependent region);                                             13
                                                                                 14
 swap(J_in, J_out);                                                              15
}                                                                                16
return_partition(J_out);                                                         17
```

Figure 3.7: Representation in pseudo-code of a nine point stencil application described at the concurrent level exploiting the *naive* method.

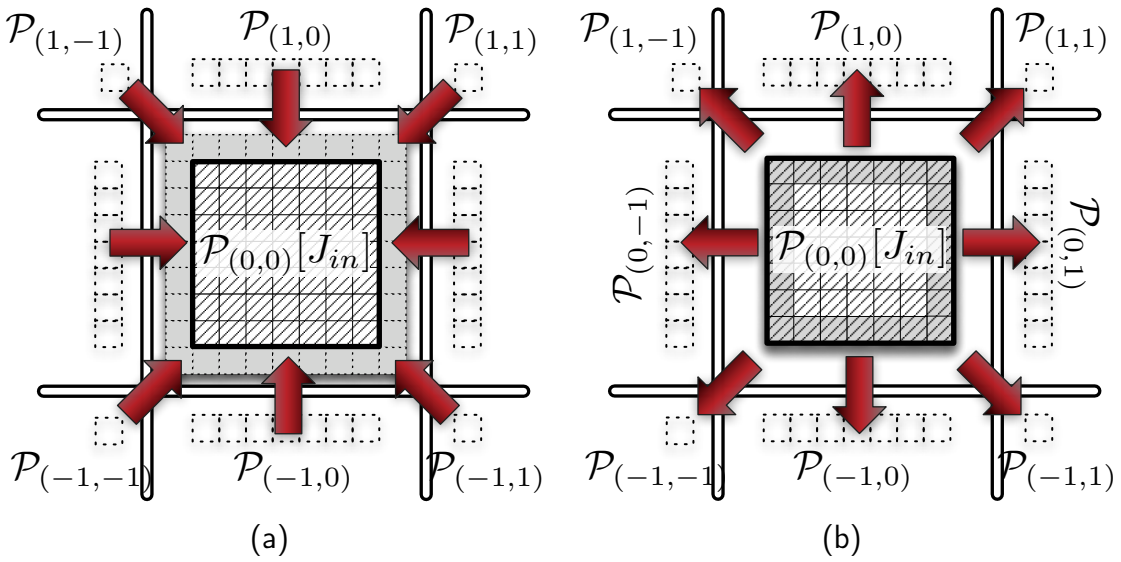


Figure 3.8: Graphical representation of the incoming (fig. 3.8(a)) and outgoing (fig. 3.8(b)) communication of a *naive* implementation of the nine point stencil at the concurrent level.

I Each process sends data to eight neighbours. We suppose communications are asynchronous with at least one degree of asynchronicity.

II The process computes the new values for the elements of the incoming independent region.

III The data required to complete the updating of elements in the incoming dependent region are acquired.

IV The values of the remaining elements are computed.

The schema of the program is simple and does not introduce any particular difficulties with regard to managing computations and communications, or when targeting architectures to support their overlaps.

The computations associated with the elements of the incoming independent region are the only ones that can possibly be overlapped with communication latency. Hence, in order to target the optimum efficiency, the region has to feature a data volume such that its computation time, i.e. the time required to update all its elements, is sufficient to completely masquerade the communication latency.

Considering that in a regular two-dimensional space, the number of possible neighbours is equal to 8, the simplest *naive* approach features 8 incoming and the same number of outgoing communications per step. Because the previous observation can be extended to spaces featuring more dimensions, we can claim the following Theorem.

Theorem 3.4.1 (*Naive* **Overhead).** *Let ψ be a generic $\mathcal{HUA}$ stencil defined over an n-dimensional spatial structure. A naive implementation of ψ at the concurrent level can require at most $3^n - 1$ incoming and $3^n - 1$ outgoing communications.*

Proof. Because in a naive implementation the partition dependencies are resolved through direct communication, we can assert the following:

$$\Delta^{in} = \Delta^{out} = \Delta^{in}_{com} = \Delta^{out}_{com}$$

Moreover, if we consider the nine point stencil, which is a worst case for communications, we can claim

$$|\Delta^{in}| = |\Delta^{out}| = |\Delta^{in}_{com}| = |\Delta^{out}_{com}| = 3^n - 1$$

□

Coming back again to our nine point stencil example, we focus on an analytic analysis of communication impact exploiting the $\mathcal{LC}$ cost model. We consider a specific configuration, where the computation time associated with the updating of a single element is equal to zero. This approach allows as to avoid the introduction of mechanisms in order to model the possible overlapping of computations and communications.

We can therefore model the communication overhead per step (T_{com}^{nine}) as follows:

$$T_{com}^{naive} = 8 * \left(t_{setup}^{snd} + t_{setup}^{rcv}\right) + t_{transm} * s_{total}^{naive}$$

The s_{total}^{naive} parameter is the total size of data that is sent by a process during a generic step. Let us suppose a square partition with l elements on each border; we have

$$s_{total}^{naive} = 4 * (l+1) size\_of(one\_el)$$

Extending the analytic analysis of communication impacts to an n-dimensional space, we can assert the following.

$$T_{com}^{naive} = (3^n - 1) * \left(t_{setup}^{snd} + t_{setup}^{rcv}\right) + t_{transm} * s_{total}^{n\_dim\_naive} \tag{3.1}$$

The $s_{total}^{n\_dim\_naive}$ parameter is the total size of data that is sent by the generic process in a generic step of the extension to n-dimensional space of the nine point stencil.

3.4.4 Concurrent Level: The $shift$ Method

An optimization of the previous *naive* schema can be introduced, exploiting Plimpton's *shift method* [46]. The strength of the approach is to avoid direct communications with diagonal neighbours; all data shift only along the main axes of the partition space.

We formally define as diagonal neighbour a neighbour partition whose associated movement vector α (see Definition 3.2.2) contains more than one non null component:

$$\alpha = (\alpha_1, \ldots, \alpha_n);\ \exists i, j\ \ \alpha_i \neq 0, \text{ and } \alpha_j \neq 0$$

In a symmetric manner, we consider non diagonal neighbours those that are placed along the main axes of the partition reference system. Therefore, the generic partition, which is associated to the movement α, does not belong to the diagonal neighbour set when:

$$\alpha = (\alpha_1, \ldots, \alpha_n);\ \exists! i\ \ \alpha_i \neq 0$$

The avoidance of diagonal communications implies that an explicit routing of the information to and from diagonal neighbours has to be implemented in the program.

In Figure 3.9, we report the pseudo-code that implements the behaviour of a generic process that executes the shift method. The program schema is quite different from the *naive* one. In terms of communications, in the *naive* case, each stencil step is characterized by a single send phase followed by a receive one. Instead, the $shift$ method features a sequence of interleaved send and receive operations. Figure 3.10(a) and Figure 3.10(b) report respectively the incoming and outgoing communication patterns. In the Figures, the arrows representing communications

```
double J_in[512][512], J_out[512][512];                                  1
load_partition_values(J_in);                                             2
for(i_step = 0; i_step < 5; i_step++){                                   3
                                                                         4
 SEND_1(P_(1,0)); SEND_1(P_(-1,0));                                      5
 COMPUTE_A(Incoming independent region);                                 6
 RECV_2(P_(1,0)); RECV_2(P_(-1,0));                                      7
                                                                         8
 SEND_3(P_(0,-1)); SEND_3(P_(0,1));                                      9
 COMPUTE_B(Incoming independent region);                                 10
 RECV_4(P_(0,-1)); RECV_4(P_(0,1));                                      11
                                                                         12
 COMPUTE(Incoming dependent region);                                     13
 swap(J_in, J_out);                                                      14
}                                                                        15
return_partition(J_out);                                                 16
```

Figure 3.9: Representation in pseudo-code of a nine point stencil application described at concurrent level exploiting the *shift* method.

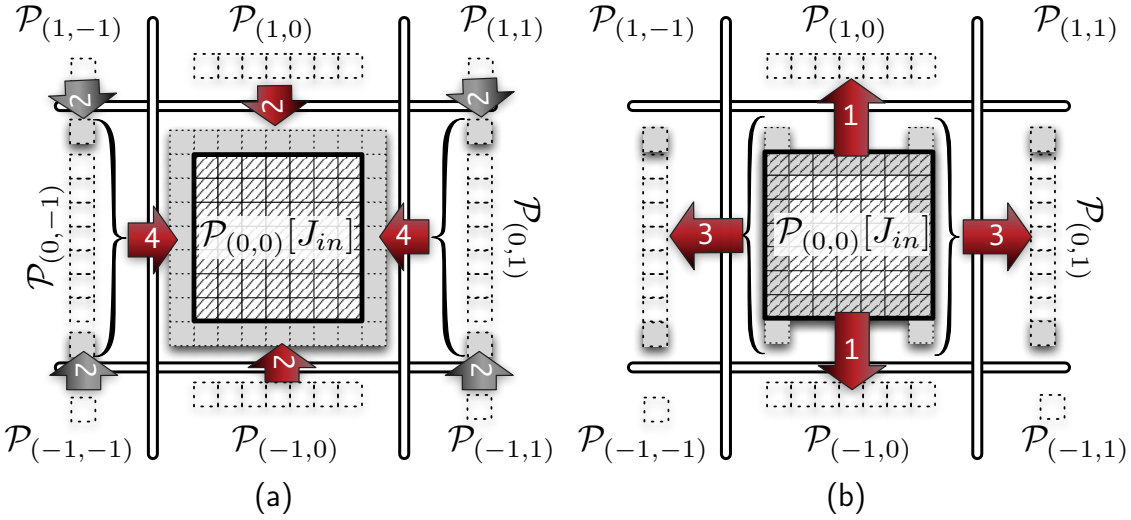


Figure 3.10: Graphical representation of the incoming (fig. 3.10(a)) and outgoing (fig. 3.10(b)) communication of a *shift* implementation of the nine point stencil at the concurrent level.

are labelled with indices that establish the temporal order according to which they are executed. The same indices are reported in the pseudo-code send and receive operations.

Considering an extension to an n-dimensional space of the nine point stencil, we can assert the following Theorem.

Theorem 3.4.2 (*Shift* **Overhead).** *Let ψ be a generic $\mathcal{HUA}$ stencil defined over an n-dimensional spatial structure. An implementation of ψ at the concurrent level requires $2 * n$ incoming and $2 * n$ outgoing communications at most.*

Proof. In an n-dimensional space, the multidimensional extension of the nine point stencil requires information exchange with all its neighbours. Exploiting Plimpton's shift methods, all direct communications with diagonal neighbours can be avoided. A process therefore communicates only with non-diagonal neighbours:

$$\Delta_{com}^{in} = \Delta_{com}^{out} = \left\{ \forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} \in \Delta^{in};\ \exists! i\ \ \beta_i \neq 0 \right\}$$

Because a component of a movement vector can be equal to -1, 0 or 1, we can assert that:

$$|\Delta_{com}^{in}| = |\Delta_{com}^{out}| = 2 * n$$

□

We now focus the discussion on the analysis of the communication impacts as we did for the *naive* method. Consider a configuration where the computation time associated with the updating of a single element is equal to zero. In this case, the communication overhead per step can be modelled as follows:

$$T_{com}^{shift} = 4 * \left(t_{setup}^{snd} + t_{setup}^{rcv} \right) + t_{transm} * s_{total}^{shift} \tag{3.2}$$

The s_{total} parameter is the total size of data that is sent by the generic process in a generic step. Instead, considering an extension to n-dimensional space of the nine point stencil, we have:

$$T_{com}^{shift} = 2 * n \left(t_{setup}^{snd} + t_{setup}^{rcv} \right) + t_{transm} * s_{total}^{n\_dim\_shift} \tag{3.3}$$

The different impacts that the two methods have on communications is evident when the previous Equation 3.3 is compared with Equation 3.1 on page 96. The two quantities $s_{total}^{n\_dim\_naive}$ and $s_{total}^{n\_dim\_shift}$ are equal: this is evident from the previous Figures in the two-dimensional case and can also be proved for a space featuring any number of dimensions. The two methods send and receive in each step the same amount of data although they are packed differently. Indeed, the *naive* method is characterized by a high number of communications which can also be of small dimensions, while the *shift* method features fewer but bigger communications.

As we can claim from the previous discussion on message sizes, the different impact of the two methods is restricted to the setup communication overhead. This

fact guarantees that the improvement of the $shift$ method will also apply when targeting $\mathcal{LC}$ implementations that support the overlapping of computations and communications.

Finally, it is worth mentioning that the weight of the performance improvement of the shift method strongly depends on the size of data sent in a single step. Indeed, if in T_{com}^{shift} the term for the setup overhead, i.e. $(2 * n) * \left(t_{setup}^{snd} + t_{setup}^{rcv}\right)$, is negligible compared to the term for the transfer overhead, i.e. $t_{transm} * s_{total}^{n\_dim\_shift}$, then the improvement of the shift method will also be negligible. Therefore the shift method is a technique that is useful when targeting fine grain computations.

3.4.5 Environment of Experimental Tests

In this Section, we describe the environment that we use to test the communication overhead of a stencil program on real architectures.

The architectures we exploited are listed below.

I **Pianosa**: a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100

II **Ottavinareale**: an eight core Intel Xeon (CPU E5420 @ 2.50GHz)

III **Siberia**: nine core Cell B.E.

On top of Siberia, we exploited our implementation of $\mathcal{LC}$ as presented in Appendix A. Because up to now we do not have any $\mathcal{LC}$ implementation for the other two architectures, we decided to target the MPI message passing library. On the Pianosa cluster, we selected the MPICH 2 version of the library. Instead, for the Ottavinareale multi-core, we selected an implementation of MPICH that is optimized for shared memory architectures (shared memory MPICH).

For each test we keep the number of partitions fixed with respect to the number of dimensions of the space. For example in a two-dimensional space we considered a partition space represented as follows:

$$[0,3] \times [0,3]$$

This is the minimum configuration such that a generic partition can feature exactly eight **distinct** neighbours. In the general case of an n-dimensional space, the smallest partition space we consider is therefore

$$\overbrace{[0,3] \times [0,3] \times \ldots \times [0,3]}^{nfactors}$$

With the number of partitions fixed, we run the tests increasing the number of elements of each partition. We used this configuration because it represents the test model which simulates the behaviours of the different implementations with different types of grain while exploiting the minimum number of resources.

3.4.6 Experimental Results

In this Section we present the results of a test implementing the *naive* and $shift$ methods. As a reference stencil-based application, we considered the nine point stencil, in a two-dimensional space, and its extension to a three-dimensional space (in this case the stencil is called the twenty seven point stencil). As for the analytic study, we focus only on the communication overhead, therefore each process in the presented test performs only communications that avoid the computation phase. As test environment, we used that described in the previous Section.

Figure 3.11 and Figure 3.12 report the performance results on the Pianosa cluster. Figure 3.13 and Figure 4.14 report the results on Ottavinareale, i.e. the Intel multi-core architecture. Finally, Figure 3.15 reports performance results on Siberia, the Cell B.E IBM multi-core architecture.

In the charts, the time gain parameter $\mathcal{G}_{com}$ is the result of dividing the step completion time of a reference method by the step completion time of a target implementation χ. For all tests, we consider the *naive* method as the reference. Formally we have:

$$\mathcal{G}_{com} = \frac{T_{com}^{naive}}{T_{com}^{\chi}}$$

From our results we can claim that, in most of the cases, the $shift$ method performs better than the *naive* method, moreover the time gain of the first is always higher for fine grains, i.e. partitions with fewer elements. Indeed, smaller partitions imply a lower size of all the transferred messages. In such a configuration, the constant setup communication overhead, which the $shift$ method aims to reduce, has a greater impact on the overall latency than the component proportional to traffic size.

Moreover, we observe that the time gain of the $shift$ method is always higher in three-dimensional space than in two-dimensional space. This aspect was also easily deducible from the presented performance model. Indeed the number of diagonal neighbours grows more than the number of non-diagonal ones with respect an increase in the number of space dimensions.

The tests which measure how the performance of the nine point stencil depends on cluster environment leads to conclusions which are not compatible with analytic studies of communication impacts. Incidentally, these results agree with ones presented by Palmer and Nieplocha [44]. These scientists conclude from their experimental studies that the optimal method between $shift$ and *naive* depends both on the architecture and the size and dimension of the problem.

Finally, it is worth pointing out the steps featured in the curves in the charts for the Intel multi-core architecture. These phenomena are observable for both *naive* and $shift$ methods and also in both two- and three-dimensional cases. Indeed, they are linked to the relation between cache line size and largest message sent. The phenomena appear when the size of this last entity exceeds the cache line size.

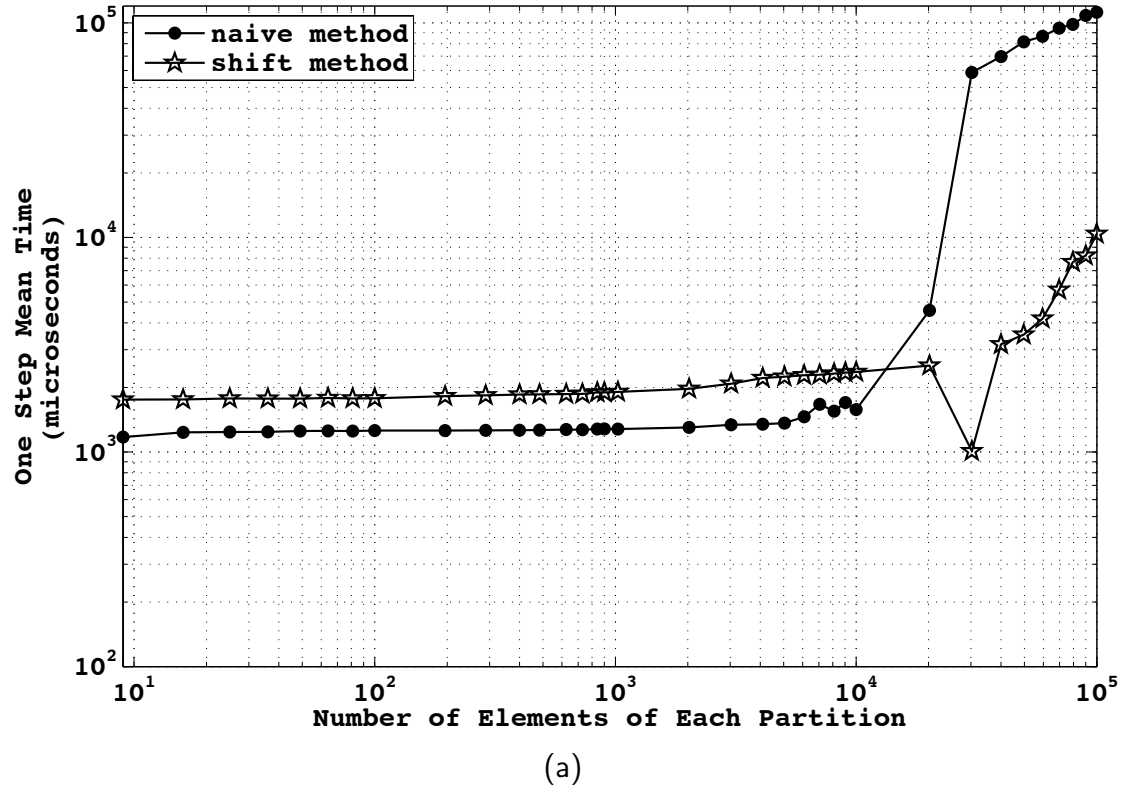
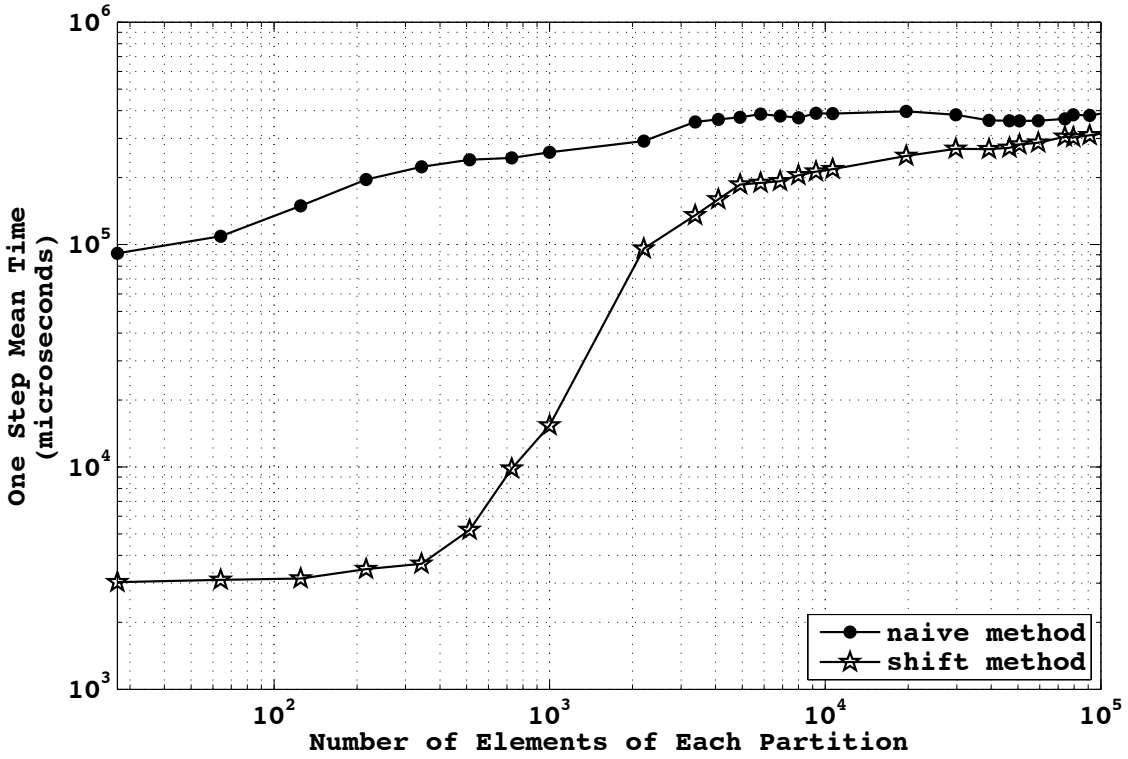
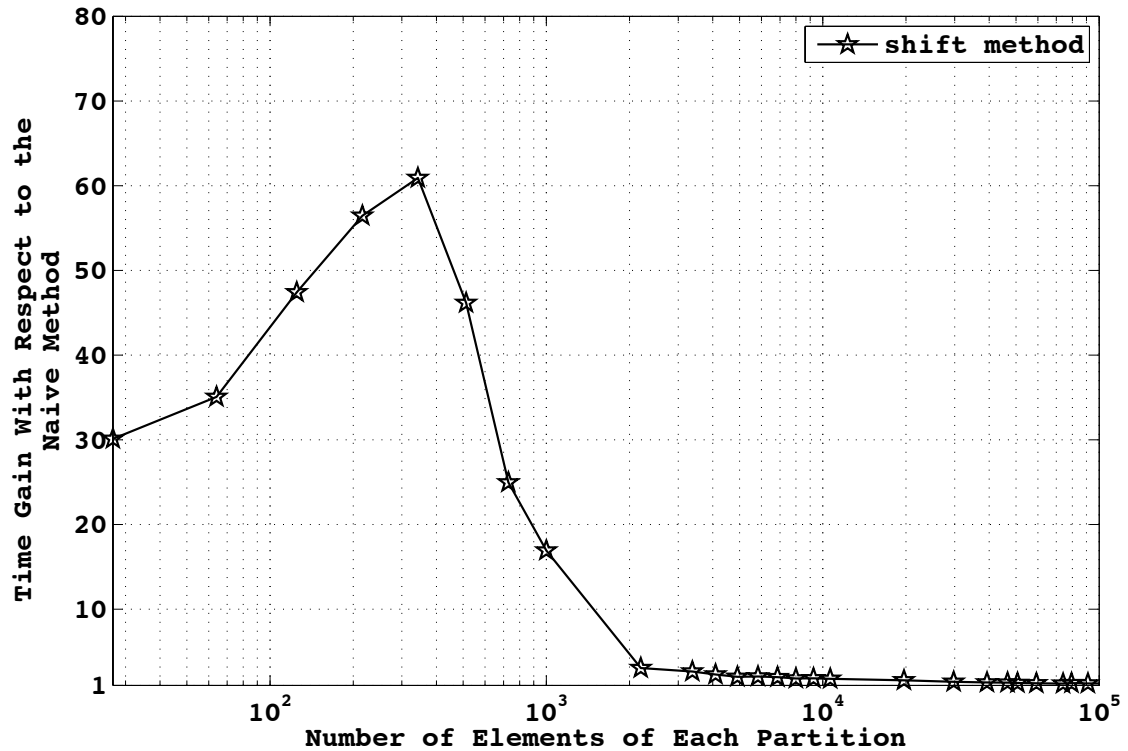


Figure 3.11: Communication overheads featured by *naive* and *shift* implementations of the nine point stencil in a two-dimensional space performed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting the MPICH library.

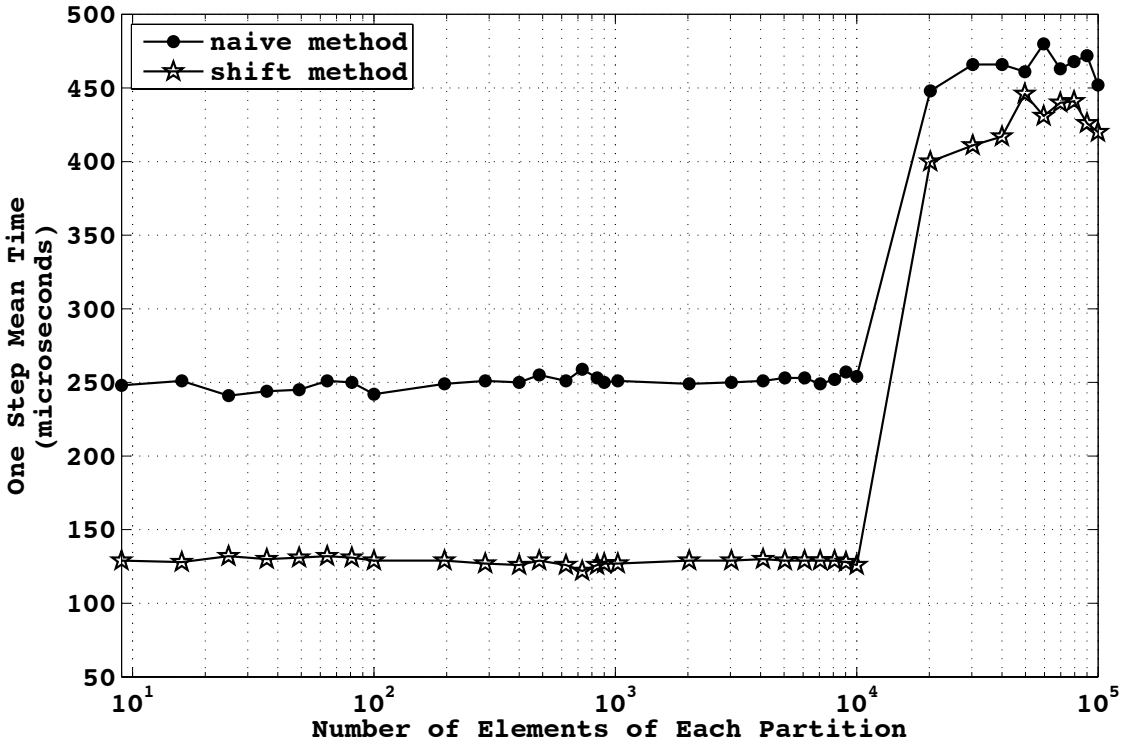


(a)

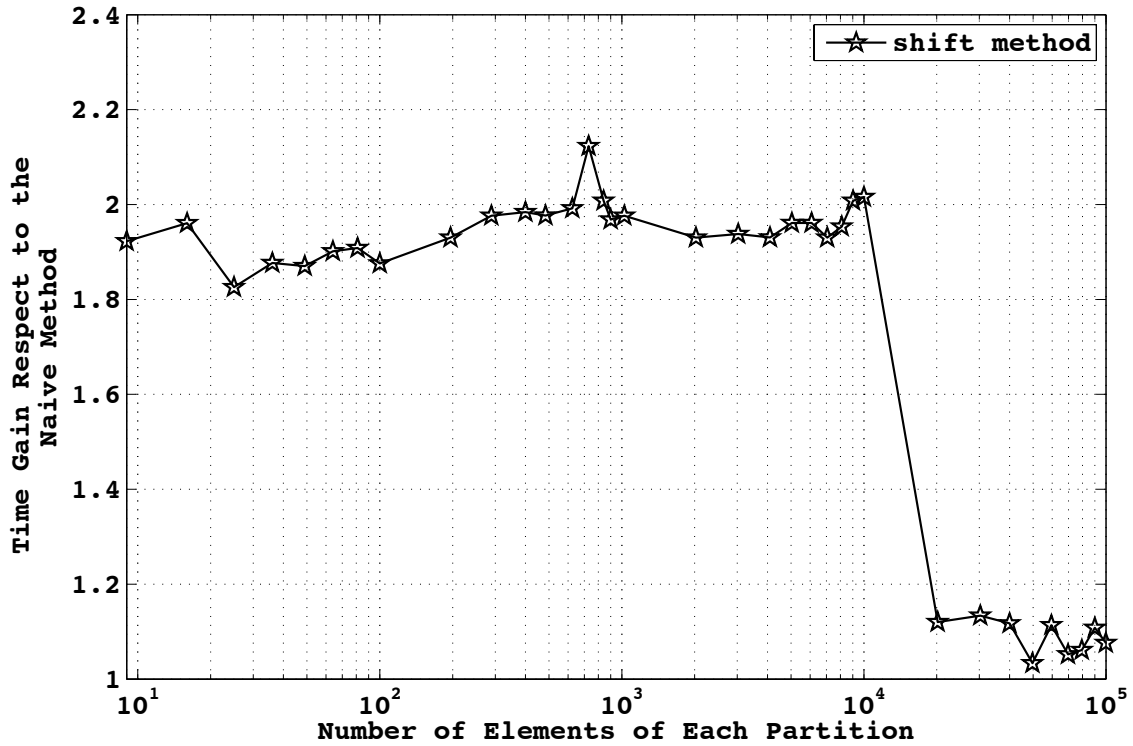


(b)

Figure 3.12: Communication overheads featured by *naive* and *shift* implementations of the twenty seven point stencil in a three-dimensional space performed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting the MPICH library.

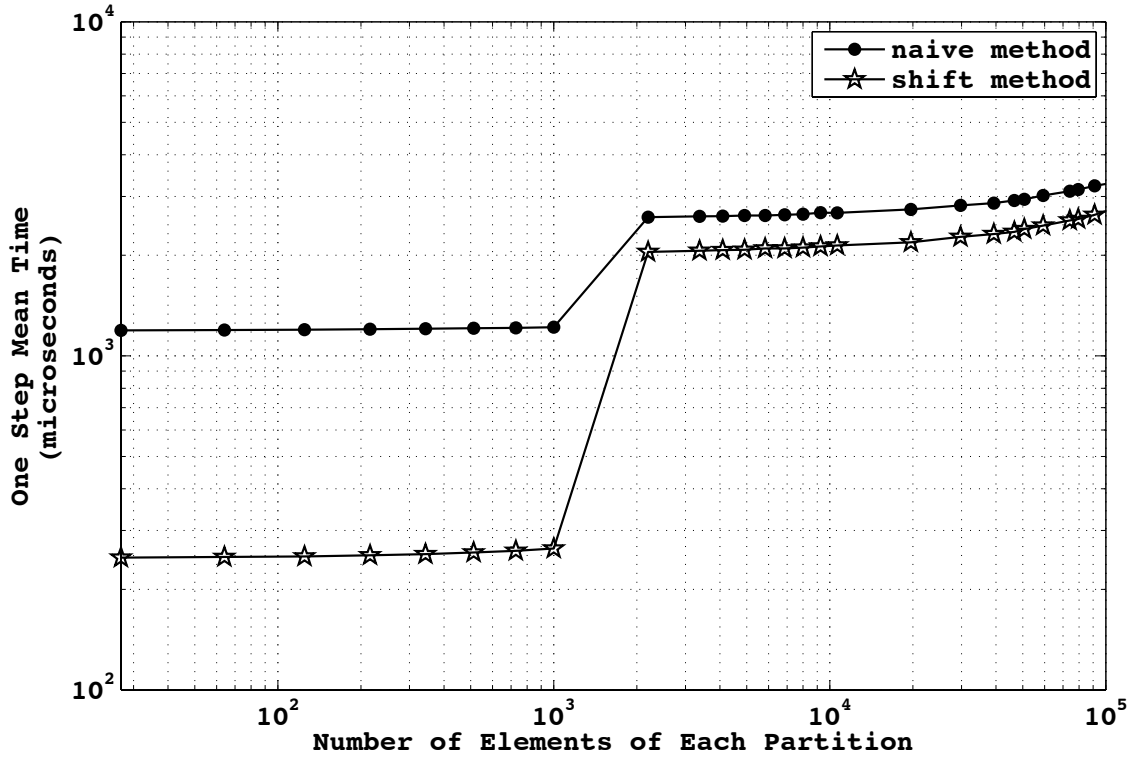


(a)

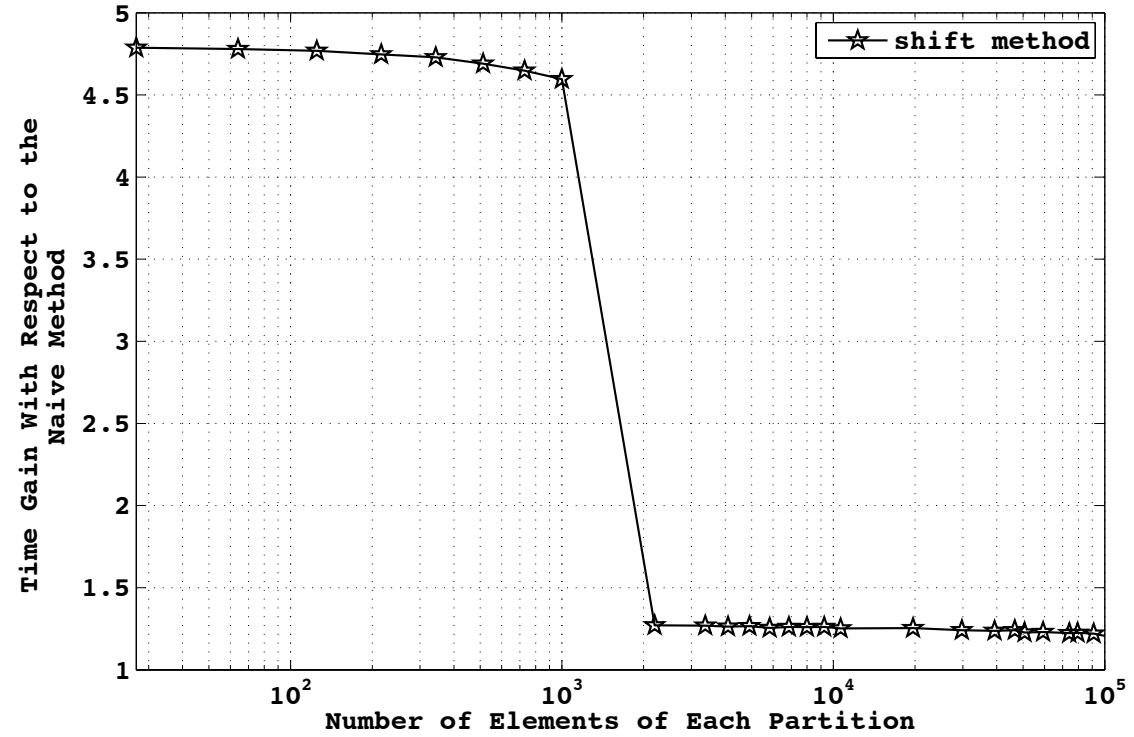


(b)

Figure 3.13: Communication overheads featured by *naive* and $shift$ implementations of the nine point stencil in a two-dimensional space performed on top of an eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting the shared memory MPICH.

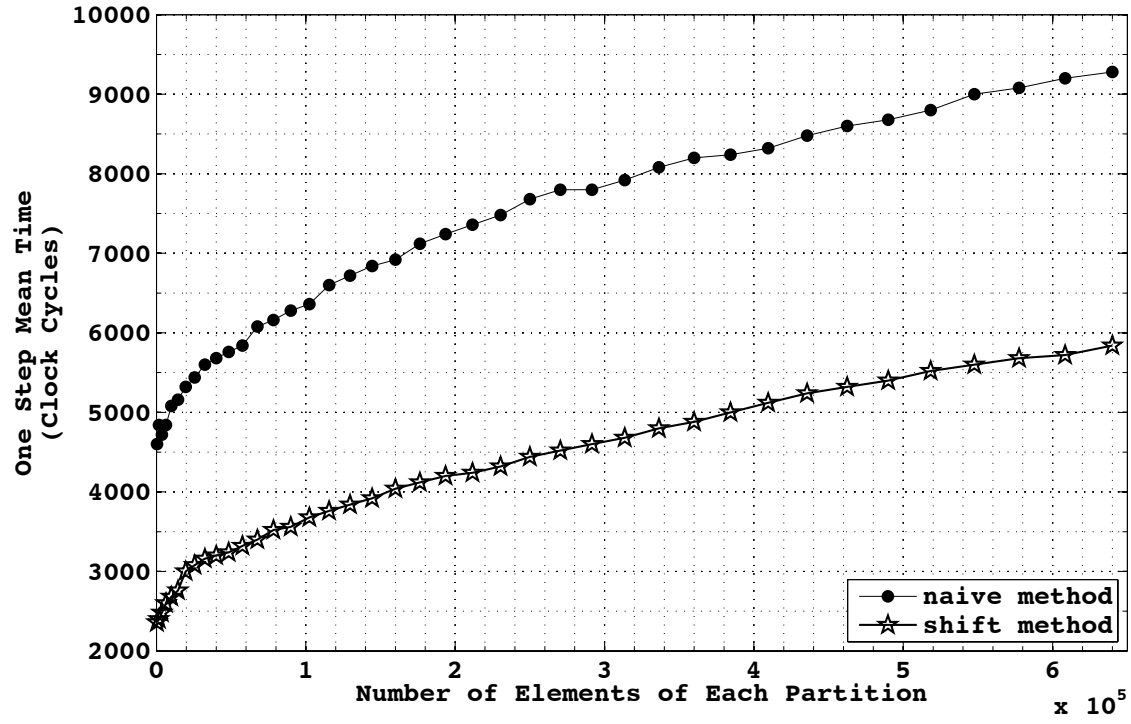


(a)

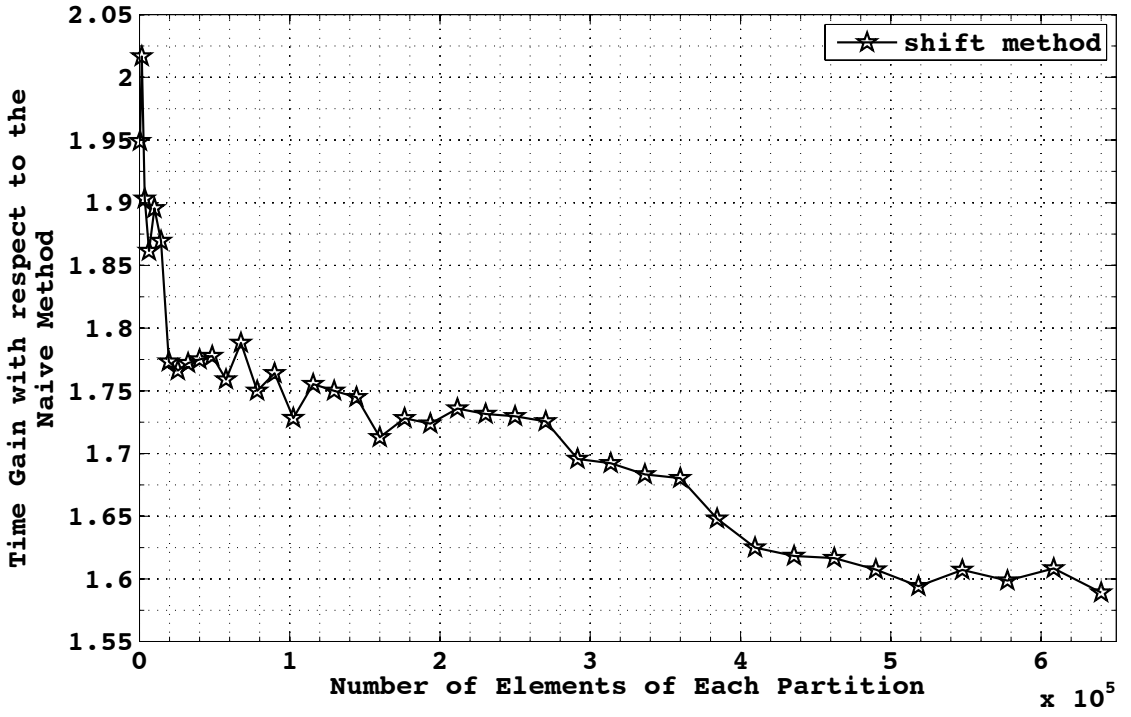


(b)

Figure 3.14: Communication overheads featured by *naive* and *shift* implementations of the twenty seven point stencil in a three-dimensional space performed on top of an eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting the shared memory MPICH.



(a)



(b)

Figure 3.15: Communication overheads featured by *naive* and $shift$ implementations of the twenty seven point stencil in a three-dimensional space performed on top of an eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting the shared memory MPICH.

Two observations support the previous assertion. Firstly, the longest messages associated with the two methods are of more or less the same size. Secondly, the steps on the curves appear in both methods at the same partition dimension.

3.4.7 Conclusions

Theorem 4.3.2 represents the best result in the literature for the minimization of the number of communications in $\mathcal{HUA}$ stencils. We wish to highlight the fact that the shift method does not introduce any benefit for example in the Jacobi or Laplace cases, where the stencil does not feature any information sharing between diagonal neighbours. Therefore, we can conclude that at the current state-of-the-art a generic $\mathcal{HUA}$ stencil requires $2 * n$ incoming and outgoing communications and secondly that the Jacobi and Laplace applications cannot be implemented with less than $2 * n$ communications. In the next Chapter we will present some optimizations which, by modifying the stencil at the functional level, break both the previous limits.

Chapter 4

$\mathcal{Q}-transformations$

Abstract

One of the techniques most used to describe stencil computations is the *owner-computes* rule. Although the rule makes the definition of stencil parallelization an easier task, we discover that alternative solutions can lead to important new optimizations that reduce the number of dependencies between domain partitions.

The new transformations, which are classified as relaxed-safe, reduce to n the maximum number of communications required to implement a $\mathcal{HUA}$ stencil, where n is the number of dimensions of the targeted space. The optimizations achieved by the new techniques represent the best results amongst solutions available in the literature.

This Chapter is structured as follows. Section 4.1 introduces $\mathcal{Q}$–$transformations$ in general and proves some of their properties.

Section 4.2 defines a particular case of $\mathcal{Q}$–$transformations$ called Positive $\mathcal{Q}-transformations$ and provides some Theorems which highlight unusual aspects of the new transformations.

Section 4.3 analyses the impact of Positive $\mathcal{Q}-transformations$ in the implementation of a nine point stencil at the concurrent level. Moreover it provides the proof that the $q-shift$ method based on Positive $\mathcal{Q}$–$transformations$ offers the best result amongst the solutions cited in literature.

Section 4.4 introduces an analysis on the impact of Positive $\mathcal{Q}$–$transformations$ in the case of stencils which do not feature partition dependencies with diagonal neighbours, such as the Jacobi stencil.

Section 4.5 presents Negative $\mathcal{Q}-transformations$ which feature opposite characteristics to their Positive counterparts. The Section demonstrates the benefit of interleaving Positive and Negative $\mathcal{Q}-transformations$ in order to reduce the overhead, rearranging if necessary the values in their original positions in the data structures.

Finally, Section 4.6 focuses on the extension of Positive and Negative $\mathcal{Q}-transformations$ to semi-uniform stencils.

Contents

4.1 $\mathcal{Q}-transformations$

Since Chapter 2, we have discussed the concept behind the owner-computes rule and, with Definition 2.1.9 on page 31, we formally revisited the rule in terms of the structured stencil model. This rule is useful for defining the functional dependencies of stencil applications, nevertheless we observed that it can preclude some important optimizations. Hence, in this Chapter we introduce $\mathcal{Q}-transformations$.

The $\mathcal{Q}-transformations$ compute modifications to a $\mathcal{HUA}$ stencil at the functional dependency level. The impact of the changes is observable at all lower levels of the system architecture (see Chapter 3).

Exploiting particular $\mathcal{Q}$–$transformations$, we will define two optimization methods that lower the maximum number of communications required to describe the stencil at the concurrent level. We already gave a preview example of this kind of $\mathcal{Q}-transformation$ in Section 2.5.2 of Chapter 3 on page 69.

4.1.1 Defining $\mathcal{Q}-transformations$

Before formally presenting $\mathcal{Q}$–$transformations$, we would like to recall and reinforce some important working hypotheses. In our studies, we focus on space invariant stencils which do not feature any *loop-carried* dependencies inside a single step. In such configurations, as we will formally prove later in this Chapter, a transformation which modifies the location of the application point of a $\mathcal{HUA}$ stencil (such an operation is equivalent to breaking the owner-computes rule) results in a relaxed-equivalent program.

We start now from a formal definition of a generic $\mathcal{Q}-transformation$.

Definition 4.1.1 (**Generic** $\mathcal{Q}-transformation$)**.** Let ψ be a $\mathcal{HUA}$ stencil. A generic $\mathcal{Q}$–$transformation$ transforms ψ into the $\mathcal{HUA}$ stencil $\mathcal{Q}[\psi]$ which is equivalent to the original one, except for the step model which is defined as follows:

$$
\begin{array}{rcl}
\forall e \in \mathcal{M} & \overset{\mathcal{Q}[\psi]}{\rightarrow} & \left(\mathcal{F}^{\psi}, \mathcal{S}^{\mathcal{Q}[\psi]}\right) \\
q & = & (q_1, \ldots, q_{dim}) \\
\mathcal{R}^{\mathcal{Q}[\psi]} & = & \mathcal{R}^{\psi} + q \\
\mathcal{S}^{\mathcal{Q}[\psi]} & = & e + \mathcal{R}^{\mathcal{Q}} \\
 & \overset{\Downarrow}{=} & e + \{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_\alpha = \beta_\alpha + q\} \\
\mathcal{M}^{i+1}_{\mathcal{Q}[\psi]}[e] & = & \mathcal{F}_i(\mathcal{M}^{i}_{\mathcal{Q}[\psi]}[e + \gamma_1], \ldots, \mathcal{M}^{i}_{\mathcal{Q}[\psi]}[e + \gamma_n])
\end{array}
$$

A generic $\mathcal{Q}$–$transformation$ introduces some modifications to the relative shape of a $\mathcal{HUA}$ stencil. The changes can be represented geometrically in two equivalent ways with respect to the selected observation point. Indeed, the changes produce a rigid translation of all the relative shape elements or equivalently they result in a translation of the application point.

Definition 4.1.1 asserts that the resulting stencil is still a $\mathcal{HUA}$ stencil. To assure the correctness of the previous definition, we report the following Property.

Property 4.1.1 ($\mathcal{Q}$*–transformations* **are** $\mathcal{HUA}$**).** *$\mathcal{Q}$–transformations map $\mathcal{HUA}$ stencils onto $\mathcal{HUA}$ stencils.*

Proof. Trivially, we know that the only difference between the original and transformed stencils is the step model. Because it is evident that the modified stencil step is compatible with the $\mathcal{HUA}$ model, we can conclude that the transformed stencil is still a $\mathcal{HUA}$ stencil. □

Another interesting characteristic of a generic $\mathcal{Q}-transformation$ is given by the following Theorem.

Theorem 4.1.1 ($\mathcal{Q}-transformation$ **relaxed safety).** *A program resulting from a general $\mathcal{Q}-transformation$ is relaxed-equivalent to the original one.*

Proof. Considering a generic domain space $\mathcal{M}$, let $\mathcal{M}_{\psi}^{i}$ be the state of the domain when applying i times a generic stencil ψ. Let $\mathcal{M}_{\mathcal{Q}}^{i}$ be the value of the domain when applying the stencil $\mathcal{Q}(\psi)$, which is the result of a generic $\mathcal{Q}-transformation$.

In order to prove the Theorem, we have to demonstrate that in a generic step i the values of $\mathcal{M}_{\mathcal{Q}}^{i}$ are the same as $\mathcal{M}_{\psi}^{i}$, apart from a space translation.

Let $\mathcal{M}^0$ be the status of the domain before applying any stencil. We can write

$$\mathcal{M}^0 = \mathcal{M}_S^0 = \mathcal{M}_Q^0 \tag{4.1}$$

If we define the element $a = e + q$, by Definition 4.1.1 on page 109, we can describe the domain state after the first step as:

$$\begin{aligned} \mathcal{M}_Q^1[e] &= \mathcal{F}_0(\mathcal{M}_Q^0[e+\gamma_1], \ldots, \mathcal{M}_Q^0[e+\gamma_n]) \\ &= \mathcal{F}_0(\mathcal{M}_Q^0[a+\beta_1], \ldots, \mathcal{M}_Q^0[a+\beta_n]) \end{aligned}$$

Therefore, for Equation (4.1), we have:

$$\mathcal{M}_Q^1[e] = \mathcal{F}_0(\mathcal{M}_S^0[a+\beta_1], \ldots, \mathcal{M}_S^0[a+\beta_n])$$

At this point, according to Definition 2.5.1 on page 68 of the $\mathcal{HUA}$ model, we arrive at:

$$\mathcal{M}_{\psi}^{1}[a] = \mathcal{M}_{\psi}^{1}[e+q] = \mathcal{M}_{Q}^{1}[e]$$

Iterating the previous reasoning for several steps, we find that:

$$\mathcal{M}_{\psi}^{i}[a] = \mathcal{M}_{\psi}^{i}[e+i*q] = \mathcal{M}_{Q}^{i}[e] \tag{4.2}$$

$$\mathcal{M}_{\psi}^{i}[e] = \mathcal{M}_{Q}^{i}[e-i*q] \tag{4.3}$$

□

Thanks to the previous Theorem, we know that we can modify a $\mathcal{HUA}$ stencil by exploiting any kind of $\mathcal{Q}-transformation$ and that the resulting stencil can be used, in place of the original one, in order to obtain the same output data values.

Up to now, the general definition of $\mathcal{Q}$–$transformations$ does not provide useful information for defining new optimization techniques. In the next Sections, we focus on two specifications of $\mathcal{Q}-transformations$ that are called positive and negative. These transformations are at the core of the optimizations that we study.

4.1.2 $\mathcal{Q}-transformations$ are different from Skewing

Before proceeding with the study of $\mathcal{Q}-transformations$, we believe that a small digression about $\mathcal{Q}-transformations$ and loop skewing is worthwhile.

Skewing [8, 24] is a well known transformation in the area of data dependency optimization theory which was invented to handle wavefront computations like the Gaus-Seidel method.

$\mathcal{Q}$–$transformations$ and Skewing transformations have in common the fact that both change the form of some geometric representation of a stencil. Incidentally, it is important to understand that the two techniques work on completely different aspects of a stencil, with different hypotheses and different aims.

A description in pseudo-code of one iteration of the Gaus-Seidel stencil in a two-dimensional space is reported in Figure 4.1, while the skewed version is reported in Figure 4.2.

```
double J[101][101];                                          1
for(x = 1; x < 100; x++){                                    2
  for(y = 1; y < 100; y++){                                  3
    J[x][y] = (J[x][y + 1] + J[x][y − 1]                     4
              +J[x + 1][y] + J[x − 1][y])/4;                 5
  }                                                          6
}                                                            7
```

Figure 4.1: Pseudo-code of one iteration of the Gauss-Seidel stencil in a two-dimensional space. The stencil features $\{(1,1),(0,1)\}$ as dependency vector and $\mathcal{R} = \{(0,1)(0,-1)(1,0)(-1,0)\}$ as shape (considering an extension of the shape element to a non $\mathcal{HUA}$ stencil)

Looking at the features of skewing transformations, we can assert the following points:

I Skewing transformations were designed for wavefront stencil applications, where each individual step features loop-dependent dependencies.

```
double J[101][101];                                                    1
for(x = 1; x < 100; x++){                                              2
  for(y = x + 1; y < 100; x + y++){                                    3
    J[x][y − x] = (J[x][y − x + 1] + J[x][y − x − 1]                   4
                  +J[x + 1][y − x] + J[x − 1][y − x])/4;               5
  }                                                                    6
}                                                                      7
```

Figure 4.2: Pseudo-code of one iteration of the skewed Gauss-Seidel stencil in a two-dimensional space. The skewed stencil features $\{(1,1),(0,1)\}$ as dependency vector and $\mathcal{R} = \{(0,1)(0,-1)(1,0)(-1,0)\}$ as shape (considering an extension of the shape element to a non $\mathcal{HUA}$ stencil)

II Skewing transformations work on the statement dependency space exploiting dependency vectors that are the main mechanisms of classical optimization theory.

III Skewing is a safe transformation, not only a relaxed one. Indeed, the outputs of the original and skewed programs are equivalent with no exception whatsoever.

IV The aim of skewing is to change the visit path that is exploited to update the data structure elements. The different path is selected to preserve data dependencies.

V The relative shape of the skewed program, which represents the functional dependency of the stencil, is equal to the shape of the original program.

While, looking at $\mathcal{Q}-transformations$, we can say the following:

I $\mathcal{Q}-transformations$ work on functional dependencies between working domain elements, represented by the shape construct, in steps that do not feature loop-dependent dependencies. Indeed, we recall that the Gaus-Seidel stencil does not belong to $\mathcal{HUA}$ stencils because, unlike the similar Jacobi stencil, it features loop-dependent dependencies in the single step computation.

II $\mathcal{Q}$–$transformations$ are relaxed-safe transformations as Theorem 4.1.1 proves. Indeed, the outputs of the original and transformed stencils are equivalent, except for a rigid translation of all the elements in the spatial structure.

III $\mathcal{Q}-transformations$ map $\mathcal{HUA}$ stencils onto $\mathcal{HUA}$ stencils as Property 4.1.1 proves. Therefore, since a transformed stencil still belongs to the $\mathcal{HUA}$ class, any strategy for the data structures visit is acceptable. Indeed, $\mathcal{HUA}$ stencils do not feature any loop-carried dependency.

IV The relative shape of the transformed stencil is completely different from the original one. Indeed, $\mathcal{Q}$–$transformations$ aim to change the functional dependencies between elements of the spatial structure with respect to the associated application point.

From the previous points, it is now clear that the two techniques are completely different.

4.2 Positive $\mathcal{Q}-transformations$

We have seen in the previous Sections that $\mathcal{Q}-transformations$ can transform a program into a relaxed-equivalent one. We are now going to focus on the definition of a set of particular $\mathcal{Q}-transformations$ which can automatically produce an equivalent relaxed program that has better performance than the original one.

4.2.1 Defining $\mathcal{Q}^+-transformations$

Definition 4.2.1 (**Positive** $\mathcal{Q}-transformation$)**.** Let ψ be a $\mathcal{HUA}$ stencil. A positive $\mathcal{Q}-transformation$ transforms ψ into the $\mathcal{HUA}$ stencil $\mathcal{Q}^+[\psi]$ which is equivalent to the original one, except for the step model that is defined as follows:

$$\begin{aligned}
\forall e \in \mathcal{M} \quad &\overset{\mathcal{Q}^+[\psi]}{\rightarrow} \quad \left(\mathcal{F}^{\psi}, \mathcal{S}^{\mathcal{Q}^+[\psi]}\right) \\
q^+ \quad &= \quad < q_1^+, \ldots, q_{dim}^+ > \\
q_i^+ \quad &= \quad -min\left\{\beta_\alpha * \epsilon_i \forall \beta_\alpha \in \mathcal{R}^{\psi}\right\} \\
\mathcal{R}^{\mathcal{Q}^+[\psi]} \quad &= \quad \mathcal{R}^{\psi} + q^+ \\
\mathcal{S}^{\mathcal{Q}^+[\psi]} \quad &= \quad e + \mathcal{R}^{\mathcal{Q}^+} \\
&\overset{\Downarrow}{=} \quad e + \left\{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_\alpha = \beta_\alpha + q^+\right\} \\
\mathcal{M}^{i+1}_{\mathcal{Q}^+[\psi]}[e] \quad &= \quad \mathcal{F}_i(\mathcal{M}^i_{\mathcal{Q}^+[\psi]}[e+\gamma_1], \ldots, \mathcal{M}^i_{\mathcal{Q}^+[\psi]}[e+\gamma_n])
\end{aligned} \tag{4.4}$$

$\epsilon = \{\epsilon_1, ...\epsilon_{dim}\}$ is the set of vectors in the natural basis of $\mathbb{N}^{dim}$ and $\beta_\alpha * \epsilon_i$ is the scalar product which returns the component of the vector β_α along the main space direction expressed by the vector ϵ_i.

The vector q^+ is called the **positive vector**.

Compared to the original stencil, only the set $\mathcal{R}$ has been changed into $Q^+(\mathcal{R})$: adding the constant positive vector q^+ to each vector element of $\mathcal{R}$. The resulting new set, i.e. $Q^+(\mathcal{R})$, features a nice property which is the fulcrum of some of the optimizations that we can derive from $\mathcal{Q}-transformations$.

Property 4.2.1 (**Shape Orientation**)**.** *The set $Q^+(\mathcal{R})$ can be represented as a set of vectors featuring only positive components:*

$$Q^+(\mathcal{R}) = \left\{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_\alpha * \epsilon_i \geqslant 0 \; \forall i \in \{1, \ldots, n\}\right\}$$

*$\epsilon = \{\epsilon_1, ...\epsilon_{dim}\}$ is the set of vectors in the natural basis of $\mathbb{N}^{dim}$, and the operator * represents the scalar products between vectors.*

Proof. By the definition of positive $\mathcal{Q}-transformations$, we have the following equation.

$$\mathcal{R}^{\mathcal{Q}^+[\psi]} = \left\{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_\alpha = \beta_\alpha + q^+ \; \forall \alpha\right\}$$

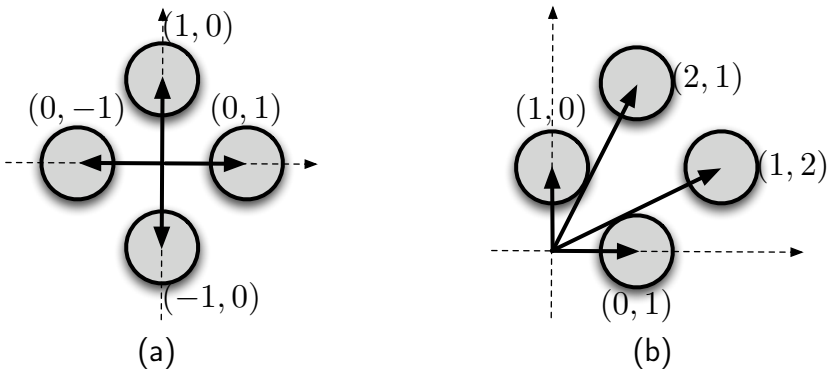


Figure 4.3: Graphical representations of the relative shapes of the $Jacobi$ and $\mathcal{Q}^{+}[Jacobi]$ stencils, respectively

```
forall((x, y) ∈ J_in){                                  1
    J_out[x][y] = (J_in[x][y + 1]                       2
    +J_in[x][y − 1] + J_in[x + 1][y]                    3
    +J_in[x − 1][y])/4;                                 4
}                                                       5
```

Figure 4.4: Jacobi pseudo-code.

Therefore the condition $\gamma_\alpha * \epsilon_i \geqslant 0$ can be rewritten as:

$$\begin{aligned}
(\beta_\alpha + q^{+}) * \epsilon_i &\geqslant 0 \\
(\beta_\alpha * \epsilon_i) + q_i^{+} &\geqslant 0 \\
(\beta_\alpha * \epsilon_i) - min\left\{(\beta_\alpha * \epsilon_i)\ \forall \beta_\alpha \in \mathcal{R}\right\} &\geqslant 0 \\
(\beta_\alpha * \epsilon_i) &\geqslant min\left\{(\beta_\alpha * \epsilon_i)\ \forall \beta_\alpha \in \mathcal{R}\right\}
\end{aligned}$$

Because the last inequality is obviously valid, we see that the formula $\gamma_\alpha * \epsilon_i \geqslant 0$ is valid as well. □

```
forall((x, y) ∈ J_in){                                  1
    J_out[x][y] = (J_in[x][y + 1]                       2
    +J_in[x + 2][y + 1] + J_in[x + 1][y]                3
    +J_in[x + 1][y + 2])/4;                             4
}                                                       5
```

Figure 4.5: $\mathcal{Q}^{+}[Jacobi]$ pseudo-code.

We consider once again the case of the Jacobi stencil. We recall that the relative shape of the stencil is $R^{Jacobi} = \{(-1,0),(1,0),(0,-1),(0,1)\}$. Positive $\mathcal{Q}-transformations$ associate the positive vector $q^+ = (+1,+1)$ to the Jacobi stencil and define the relative shape of the stencil $\mathcal{Q}^+[Jacobi]$ as follows:

$$\mathcal{R}^{\mathcal{Q}^+[Jacobi]} = \big\{(0,1),(2,1),(1,0),(1,2)\big\}$$

All components of the new relative shape are not negative, as we expected from the previous Property. As we will show in the next Section, one of the effects of this geometric feature will be a kind of "*orientation*" of communications.

A graphical comparison of both the original and transformed relative shapes is shown in Figure 4.3. The shapes are geometrically the same, except for the location of the application point which is modelled by the reference system origin.

We also report the pseudo-codes of both the original Jacobi and $\mathcal{Q}^+(Jacobi)$, respectively, in Figure 4.22 and Figure 4.5.

4.3 A Nine Point Stencil at Work with $\mathcal{Q}^+ - transformations$

In this Section we come back to the nine point stencil, which we used in the previous Chapter as a worst case for communication analysis, in order to study the benefit of exploiting positive $\mathcal{Q} - transformations$.

4.3.1 Functional Dependency Level

As we have seen, $\mathcal{Q} - transformations$ introduce changes in the original stencil that are expressed directly at the functional dependency level. The result of a $\mathcal{Q} - transformation$ is a $\mathcal{HUA}$ stencil. We report the step model of the stencil $\mathcal{Q}^+[nine]$ in Table 4.1 on page 118. The stencil description can be compared with the description of the original stencil reported in the previous Chapter in Table 3.1 on page 93. Finally, a graphical representation of the new stencil shape is shown in Figure 4.6(a).

4.3.2 Partition Dependency Level

The benefit of exploiting $\mathcal{Q} - transformations$ becomes clear when expressing the $\mathcal{HUA}$ stencil description at the partition dependency level. Figure 4.6(b) and Figure 4.6(c) highlight all the regions of which the generic partition is composed.

Because the vectors in the relative shape do not have negative components, the elements of both the incoming and outgoing dependent regions are concentrated only over two edges of the partition borders.

As a consequence of the previous observation, we see that the incoming and outgoing partition sets are strict subsets of the neighbouring partition set:

$$\Delta^{in} \subset NS(\mathcal{P}_\alpha)$$

$$\Delta^{out} \subset NS(\mathcal{P}_\alpha)$$

Extending the previous assertion to a generic stencil we can define the following Property.

Property 4.3.1 (Partition Dependency Reduction with $\mathcal{Q}$–$transformations$). *Let ψ be a $\mathcal{HUA}$ stencil and $\mathcal{Q}^+[\psi]$ the stencil resulting from the application of a positive $\mathcal{Q}$–$transformation$ to ψ. We can assert the following relations between the sets of the transformed stencil:*

$$\Delta^{in}_{\mathcal{Q}^+[\psi]} \subset NS(\mathcal{P}_\alpha)$$

$\mathcal{HUA}$ STEP model of $\mathcal{Q}^{+}[nine]$

$$
\begin{array}{rcl}
\forall e=(x,y)\in\mathcal{M}_{\mathcal{Q}^{+}[nine]} & \overset{\mathcal{Q}^{+}[nine]}{\rightarrow} & \Big(\mathcal{F},\mathcal{S}_{ie}^{\mathcal{Q}^{+}[nine]}\Big)\\
\mathcal{R}^{\mathcal{Q}^{+}[nine]} & = & \Big\{(x,y+1),(x,y-1),(x+1,y-1),\\
 & & (x+1,y),(x+1,y+1),(x-1,y-1),\\
 & & (x-1,y),(x-1,y+1)\Big\}+\big(+1,+1\big)\\
 & = & \Big\{(x+1,y+2),(x+1,y),(x+2,y),\\
 & & (x+2,y+1),(x+2,y+2),(x,y),\\
 & & (x,y+1),(x,y+2)\Big\}\\
\mathcal{S}_{(x,y)}^{\mathcal{Q}^{+}[nine]} & = & (x,y)+\mathcal{R}^{\mathcal{Q}^{+}[nine]}\\
\mathcal{F} & : & \mathbb{R}^{4}\mapsto\mathbb{R}\\
\mathcal{F} & : & \big(r_1,r_2,r_3,r_4\big)\mapsto\dfrac{\sum_{i=1}^{8}r_i}{8}\\
\mathcal{M}_{\mathcal{Q}^{+}}^{i+1}[e] & = & \mathcal{F}\big(\mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x+1,y+2)],\mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x+1,y)],\\
 & & \mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x+2,y+2)],\mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x+2,y+1)],\\
 & & \mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x+1,y)],\mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x,y+2)],\\
 & & \mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x,y+1)],\mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x,y)],\\
 & & \mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x+2,y+1)],\mathcal{M}_{\mathcal{Q}^{+}}^{i}[(x,y+1)]\big)
\end{array}
$$

Table 4.1: Step component in the structured stencil model of the $\mathcal{Q}^{+}[nine]$ stencil application.

$$\Delta^{out}_{\mathcal{Q}^+[\psi]} \subset NS(\mathcal{P}_\alpha)$$

$\mathcal{P}_\alpha$ is a generic partition in which the working domain has been divided and NS is the set of neighbouring partitions of $\mathcal{P}_\alpha$.

Proof. Exploiting Property 4.2.1 about the shape orientation introduced by $\mathcal{Q} - transformations$, we can define the incoming and outgoing partition dependency set as follows:

$$\Delta^{in} = \left\{ \forall \mathcal{P}_{(x,y)} \in NS(\mathcal{P}_\alpha) \mid x \geq 0, y \geq 0 \right\}$$

$$\Delta^{out} = \left\{ \forall \mathcal{P}_{(x,y)} \in NS(\mathcal{P}_\alpha) \mid x \leq 0, y \leq 0 \right\}$$

Because all the neighbouring partitions which feature an index with negative components do not belong to Δ^{in}, the set is a strict subset of $NS(\mathcal{P}_\alpha)$.

By symmetry, we can make the same assertion about the outgoing partition dependency set. □

4.3.3 Concurrent Level: The q Method

What we previously called the effect of the "*orientation*" of communications can be analyzed at the concurrent level.

Given that $\mathcal{Q}^+[nine]$ is a $\mathcal{HUA}$ stencil, we recall that in the previous Chapter we showed two different methods for implementing the exchange of information which is needed to resolve the partition dependencies.

We first consider the *naive* method which resolves incoming and outgoing partition dependencies by direct communication between processes. Each process, which manages its own partition of the spatial structure, has to send data to the processes belonging to the outgoing partition set (Δ^{out}) and receive data from those belonging to the incoming partition set (Δ^{in}).

We introduce a new nomenclature; in order to highlight that the *naive* method is applied to a stencil which is the result of $\mathcal{Q} - transformations$, we rename the *naive* method as a q method.

We report in Figure 4.7 the pseudo-code representing the behaviour of the generic process that implements the q method for the nine point stencil application. As we did in the previous Chapter, we have used the name of partitions in the code to indicate the associated processes. Moreover, for the sake of simplicity, in communication operations, we just indicate the sender or the receiver process, while we leave to both Figure 4.8(a) and Figure 4.8(b) the burden of graphically highlighting the exact elements involved in communications.

Each step in the q method is divided into the following phases:

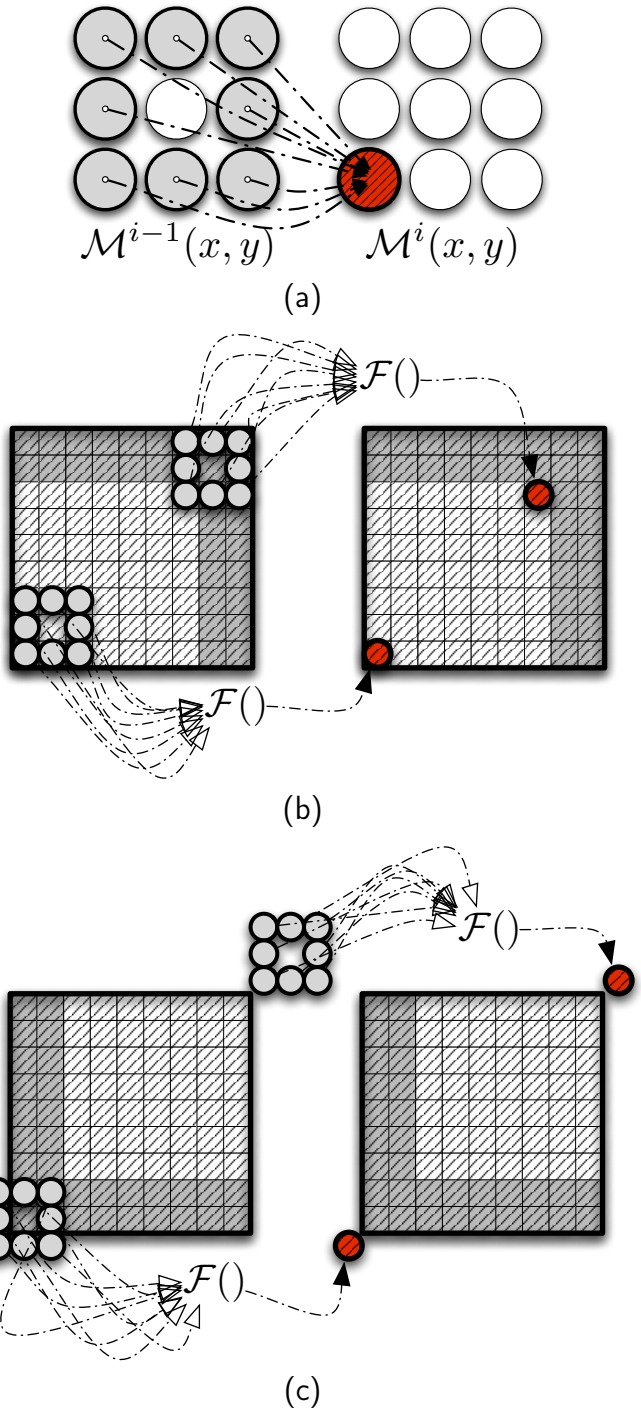


Figure 4.6: Graphical representation of stencil shape (4.6(a)), incoming dependent (colored in gray) and independent regions (4.6(b)), and finally outgoing dependent (colored in gray) and independent regions (4.6(c)) for the $\mathcal{Q}^{+}[nine]$ stencil.

```
double J_in[512][512], J_out[512][512];                                  1
load_partition_values(J_in);                                             2
for(i_step = 0; i_step < 5; i_step++){                                   3
                                                                         4
 SEND(P_(0,1)); SEND(P_(+1,0)); SEND(P_(+1,+1));                         5
                                                                         6
 COMPUTE(Incoming independent region);                                   7
                                                                         8
 RECV(P_(0,1)); RECV(P_(+1,0)); RECV(P_(+1,+1));                         9
                                                                         10
 COMPUTE(Incoming dependent region);                                     11
                                                                         12
 swap(J_in, J_out);                                                      13
}                                                                        14
return_partition(J_out);                                                 15
```

Figure 4.7: Representation in pseudo-code of a $\mathcal{Q}^+[nine]$ stencil described at the concurrent level exploiting the q method.

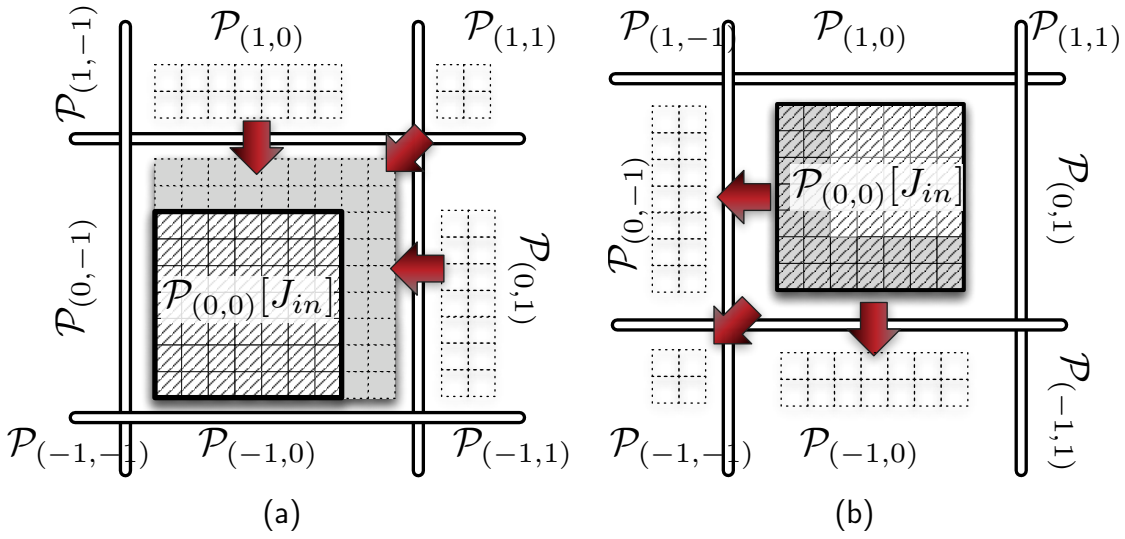


Figure 4.8: Graphical representation of the incoming (fig. 4.8(a)) and outgoing (fig. 4.8(b)) communications of a q-implementation of the nine point stencil at the concurrent level.

I Each process sends data to the three neighbours that are associated with a movement vector with non negative components. We assume the communications feature has at least one degree of asynchronicity; otherwise the presented program would stall in a deadlock situation.

II The process computes the new values for the elements of the incoming independent region.

III The data required to complete the updating of elements of the incoming dependent region are acquired from the three neighbour partitions that are associated with a movement vector featuring non positive components.

IV The values of the remaining elements are computed.

As for the *naive* method, the schema of the program is simple and does not introduce any particular difficulties with managing either computations or communications, or when supporting overlapping in target architectures.

For the nine point stencil, which represents our reference worst case, we can assert that the q method requires only three incoming and three outgoing communications. Therefore, the new q method reduces the number of communications compared to both *naive* and $shift$ methods. Up to now, the q method provides the best optimizations technique in a two-dimensional space.

The following Theorem extends the analysis to a space with a generic number of dimensions.

Theorem 4.3.1 (**The Impact of the q Method on Communications**). *Let ψ be a generic $\mathcal{HUA}$ stencil defined over an n-dimensional spatial structure. A q implementation of ψ at the concurrent level can require at most 2^n-1 incoming and 2^n-1 outgoing communications.*

Proof. Because partition dependencies are resolved by direct communications, the number of incoming communications is equal to the cardinality of the incoming partition set:

$$\Delta^{in}_{com} = \Delta^{in} = \left\{\forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} \in NS(\mathcal{P}_\alpha) | \forall \beta_i, \beta_i \geq 0\right\}$$

Because in the definition of neighbour index set (see Definition 3.2.2 on page 82) we know that $\beta_i \in \{-1,0,1\}$ and that $(\beta_1,\ldots,\beta_n) \neq (0,\ldots,0)$, we rewrite the previous formula of the incoming partition set as:

$$\Delta^{in}_{com} = \Delta^{in} = \left\{\forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} \in NS(\mathcal{P}_\alpha) | \forall \beta_i, \in \{0,1\} and (\beta_1,\ldots,\beta_n) \neq (0,\ldots,0)\right\}$$

The cardinality of the set is equal, except for having one unit less, to one of the sets composed of all possible strings of length n that can be created from an alphabet with two digits: 2^n. We can finally assert the following:

$$\left|\Delta^{in}_{com}\right| = \left|\Delta^{in}\right| = 2^n - 1$$

The same reasoning can be applied to the outgoing partition set whose cardinality is equal to the number of outgoing communications.

$$\left|\Delta^{out}_{com}\right| = \left|\Delta^{out}\right| = 2^n - 1$$

□

In Figure 4.11, we provide a chart which represents, as a function of the number of space dimensions, the number of incoming communications for each of the methods studied: *naive*, $shift$ and q. What we can see is that the new q method provides a lower number of communications than the $shift$ method only when the number of space dimensions is less than two. Therefore, the gain of the q method is limited to spaces with a small number of dimensions. We improve the results achieved with another new method introduced in the following Section.

4.3.4 Concurrent Level: The q_shift method

Like the *naive* method, the previous q method features communications with diagonal neighbours. These can be avoided by exploiting Plimpton's $shift$ method. We call the mix of q and $shift$ methods q_shift. The new method resolves the partition dependencies that the $\mathcal{Q}^+[nine]$ stencil implies with diagonal neighbours exploiting indirect communications.

In Figure 4.9, we report the pseudo-code that implements the behaviour of a generic process that executes the $q - shift$ method. The program features a pair of interleaved send and receive operations. Figure 4.10(a) and Figure 4.10(b) report respectively the incoming and outgoing communication patterns. In the Figures, the arrows representing communications are labelled with indices that establish the temporal order according to which they are executed. The same indices are exploited in the pseudo-code to tag the corresponding send and receive operations.

We can assert that the implementation of the nine point stencil that exploits the q_shift method requires only two incoming and two outgoing communications. In the two-dimensional space, the new method provides the best result amongst all other methods

The following Theorem extends the result to spaces with a general number of dimensions.

Theorem 4.3.2 (**The Impact of the q_shift Method on Communications).** *Let ψ be a generic $\mathcal{HUA}$ stencil defined over an n-dimensional spatial structure. By exploiting the q_shift method, an implementation of ψ at the concurrent level can require at most n incoming and n outgoing communications.*

```
double J_in[512][512], J_out[512][512];                                  1
load_partition_values(J_in);                                             2
for(i_step = 0; i_step < 5; i_step++){                                   3
                                                                         4
 SEND_1(P_(1,0));                                                        5
 COMPUTE_A(Incoming independent region);                                 6
 RECV_2(P_(1,0));                                                        7
                                                                         8
 SEND_3(P_(0,1));                                                        9
 COMPUTE_B(Incoming independent region);                                 10
 RECV_4(P_(0,1));                                                        11
                                                                         12
 COMPUTE(Incoming dependent region);                                     13
 swap(J_in, J_out);                                                      14
}                                                                        15
return_partition(J_out);                                                 16
```

Figure 4.9: Representation in pseudo-code of a $\mathcal{Q}^{+}[nine]$ stencil described at the concurrent level exploiting the $q - shift$ method.

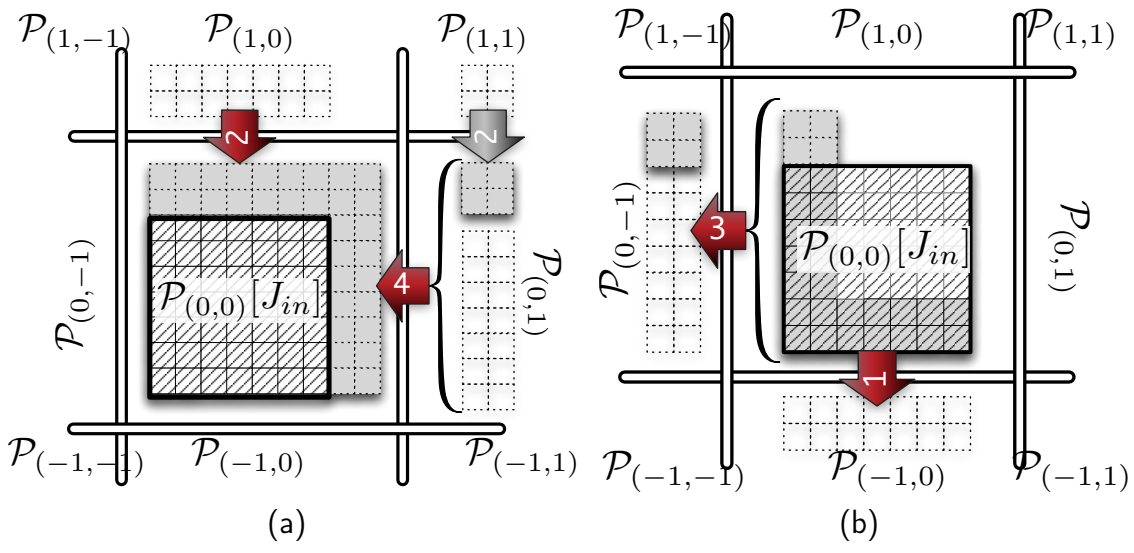


Figure 4.10: Graphical representation of the incoming (fig. 4.10(a)) and outgoing (fig. 4.10(b)) communications of a $q - shift$ implementation of the nine point stencil at the concurrent level.

Proof. In order to prove the Theorem, we consider ψ the multidimensional extension of the *nine* stencil. Indeed, we proved that *nine* is a worst case stencil for the number of communications.

The partition dependencies featured by the stencil are represented by the incoming and outgoing partition sets:

$$\Delta^{in}(\mathcal{P}_\alpha) = \left\{\forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} \in NS(\mathcal{P}_\alpha) | \forall i, \beta_i \geq 0\right\}$$

$$\Delta^{out}(\mathcal{P}_\alpha) = \left\{\forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} \in NS(\mathcal{P}_\alpha) | \forall i, \beta_i \leq 0\right\}$$

At the concurrent level, by exploiting Plimpton's shift methods, partition dependencies with diagonal neighbours are resolved without direct communication. Therefore, $\Delta^{com\_in}(\mathcal{P}_\alpha)$, the set of processes that feature send operation where $\mathcal{P}_\alpha$ is the receiver, can be modelled as follows.

$$\Delta^{com\_in}(\mathcal{P}_\alpha) = \Delta^{in}(\mathcal{P}_\alpha) \cap \left\{\forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} \in NS(\mathcal{P}_\alpha) | \exists!\ i\ \beta_i \neq 0\right\}$$

We can rewrite the previous formula as:

$$\begin{aligned}\Delta^{com\_in}(\mathcal{P}_\alpha) \quad = \quad & \left\{\forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} \in NS(\mathcal{P}_\alpha) | \exists!\ i\ \beta_i \neq 0 and\ \beta_i \geq 0\right\} \\ & \left\{\forall \mathcal{P}_{(\beta_1,\ldots,\beta_n)} | \forall j\ \beta_j \in \{0,1\} and\ \exists!\ i\ \beta_i \neq 0\right\}\end{aligned}$$

For the number of incoming communications, we easily obtain the following formula:

$$\left|\Delta^{com\_in}\right| = n$$

The same reasoning can be applied to outgoing communications, in order to assert the following result:

$$\left|\Delta^{com\_out}\right| = n$$

□

4.3.5 Experimental Results

We considered a nine point stencil and its extension to a three-dimensional space (twenty seven point stencil) as tests for comparing all four presented methods: *naive*, $shift$, q, and q_shift.

For all the tests we used the environment presented in Section 3.4.5. As for the previous tests of the *naive* and $shift$ methods, in order to focus only on the

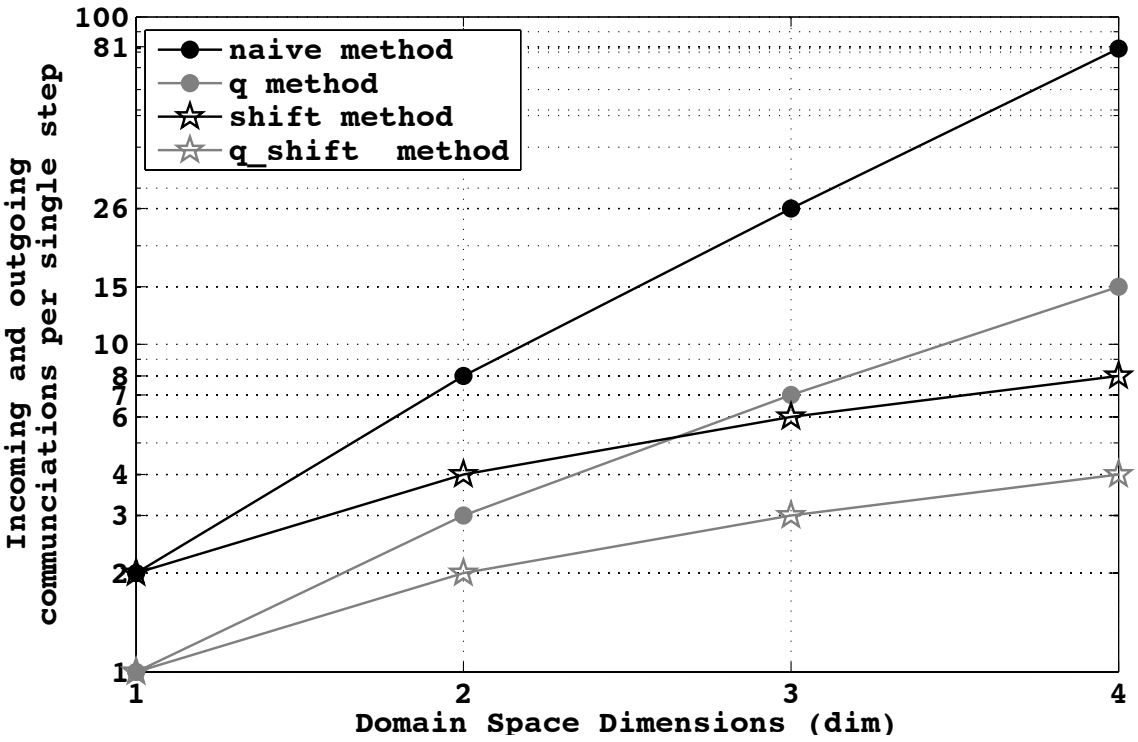


Figure 4.11: Number of communications per step exploiting different methods

communication overhead we do not consider any computations associated with each element of the spatial structure.

The charts in Figure 4.12(a) and Figure 4.13(a) report the results of the tests run on the Intel multi-core architecture. It is interesting to observe that on this chip-multiprocessor architecture, featuring intra-chip communications, the results agree closely with the forecasts that can be extracted from the chart in Figure 4.11. The chart reports, for each method and for a set of space dimensions, the number of communications.

Indeed, the measured performances are characterized by the following invariant: the method which features a lower number of communications also supports a lower communication overhead. This observation is stressed by the comparison of behaviours of the $shift$ and q methods in the two-dimensional and three-dimensional cases.

In two dimensions, the q method uses fewer communications than the $shift$ methods and this is also reflected by the performance test where q has a lower overhead. When we go to analyse the three-dimensional case, the difference in the number of communications is inverted as is the performance trend: the $shift$ method performs a step iteration faster than q.

A comment has to be made about the steps that appear in the curves for all methods in both performance charts. The causes are caching effects during the message copying phase as mentioned in the discussion on the performance of the

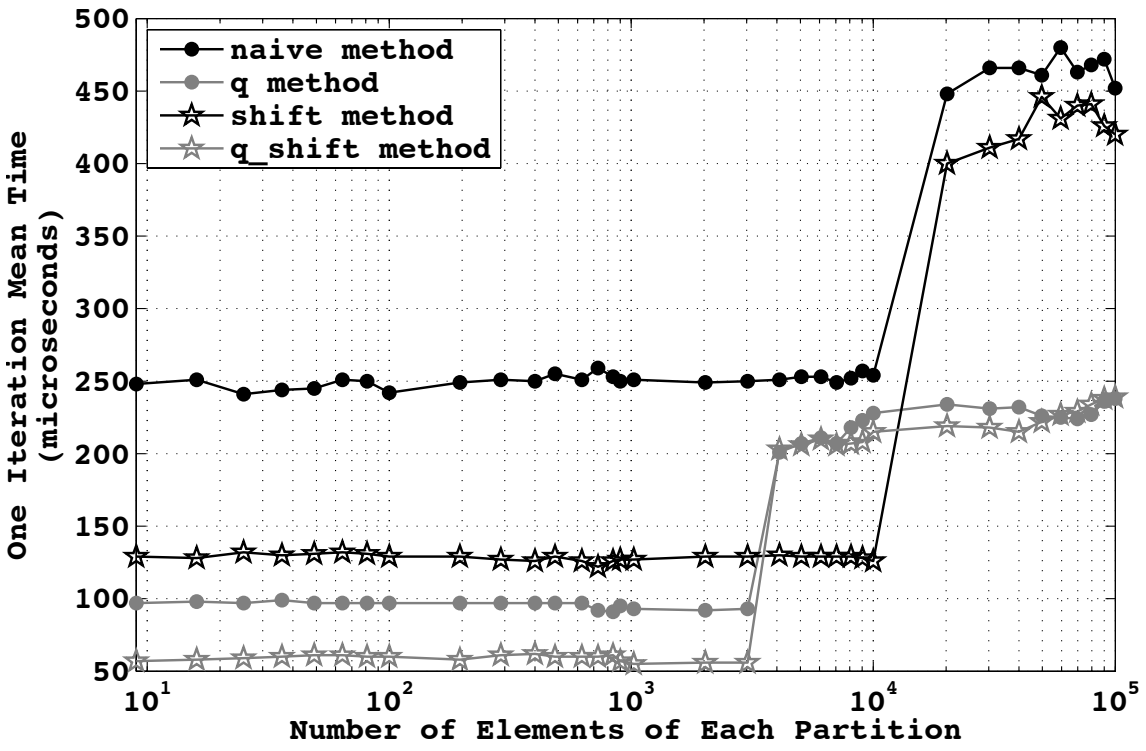
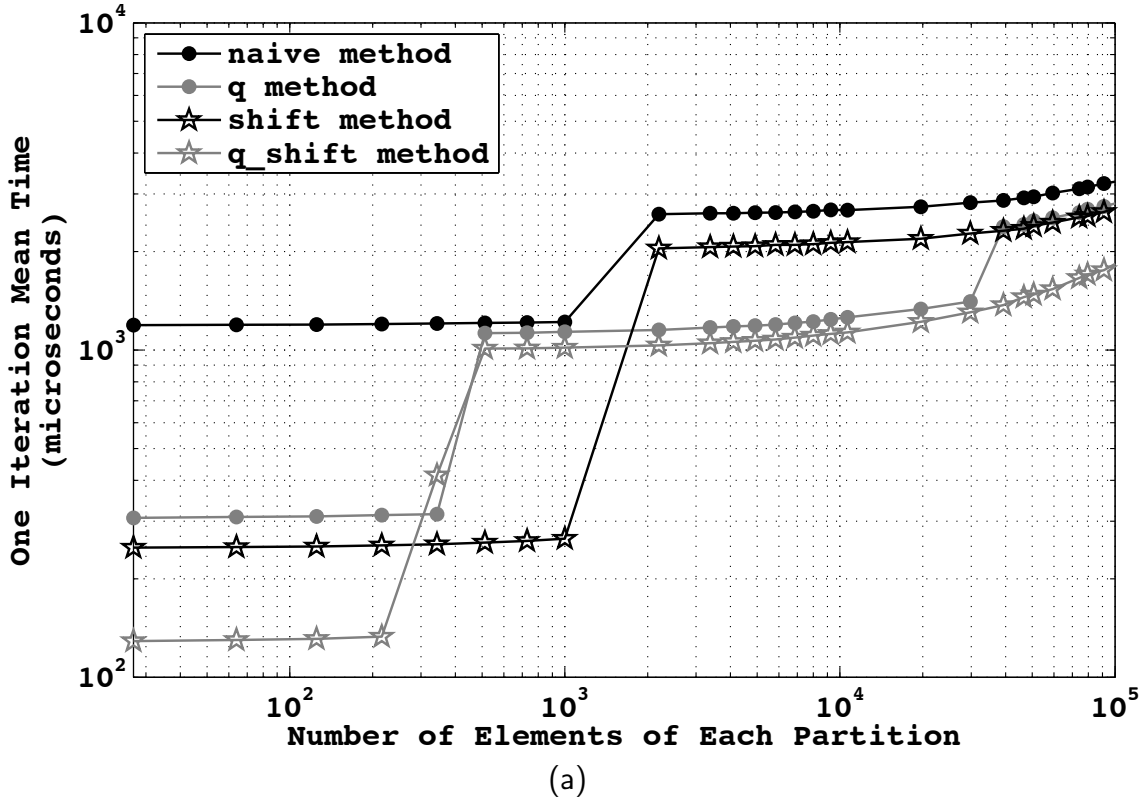
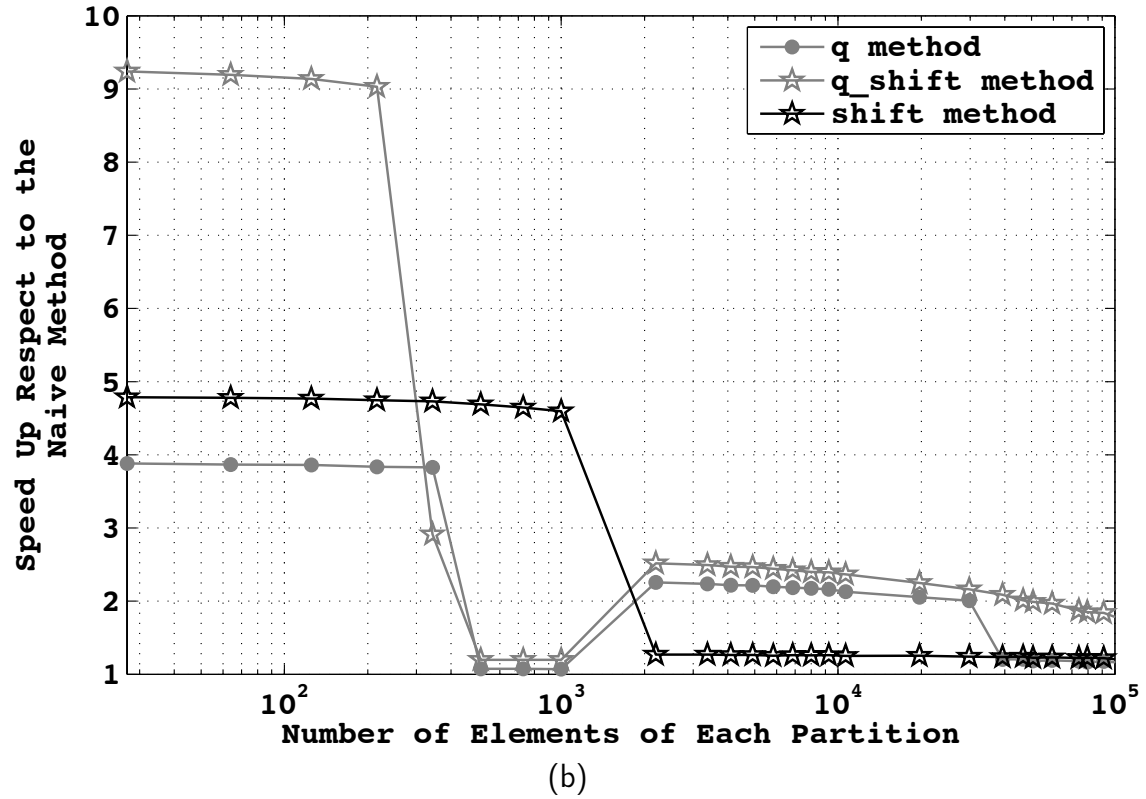


Figure 4.12: Performance results for a nine point stencil in a two-dimensional space, executed on an eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.

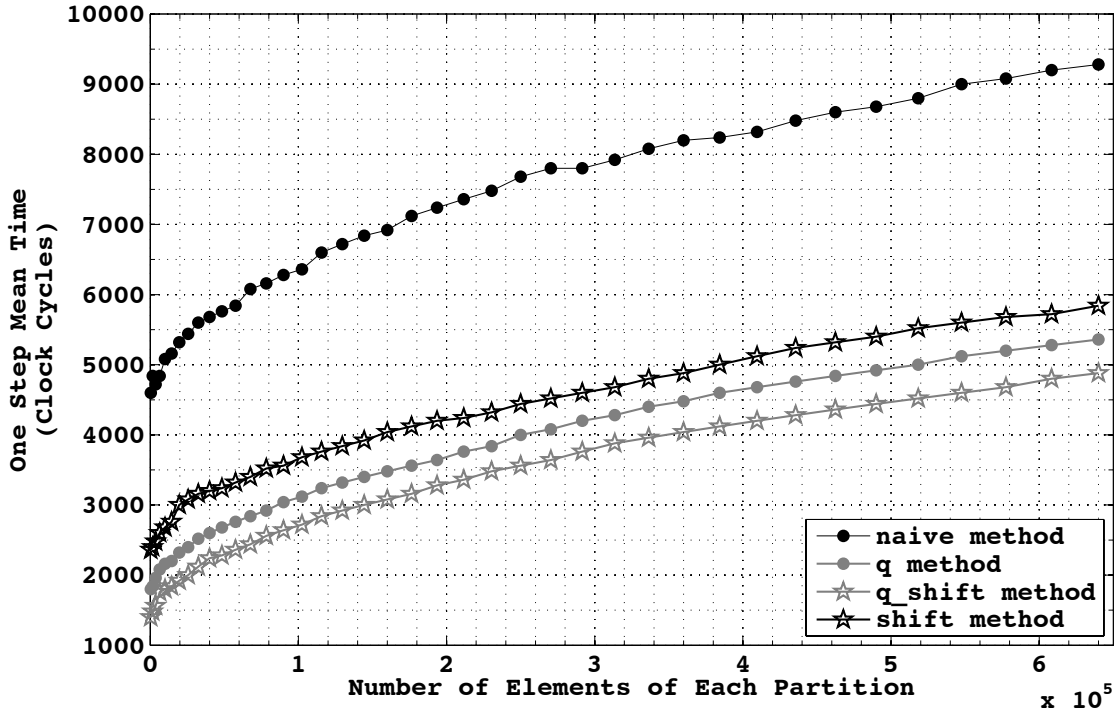


(a)

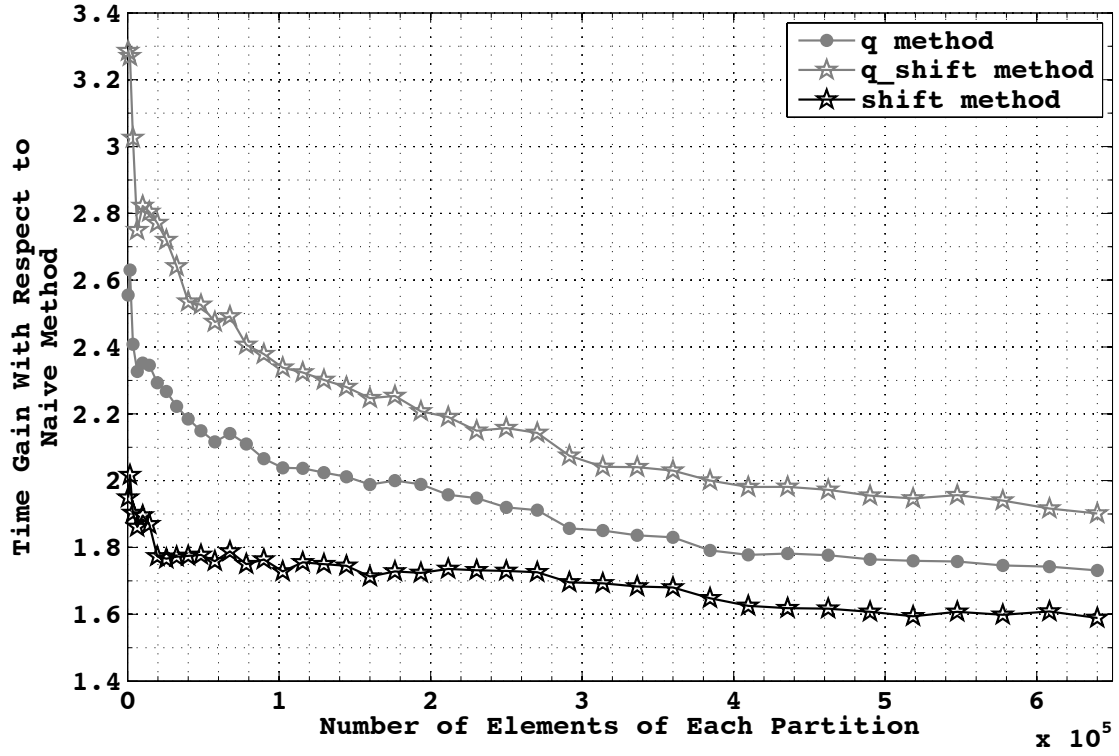


(b)

Figure 4.13: Performance results for a twenty seven point stencil in a three-dimensional space, executed on an eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.



(a)



(b)

Figure 4.14: Nine point stencil in a two-dimensional space, performed on a Cell B.E. IBM multicore exploiting the $\mathcal{M}ammu\mathcal{T}$ implementation of $\mathcal{LC}$.

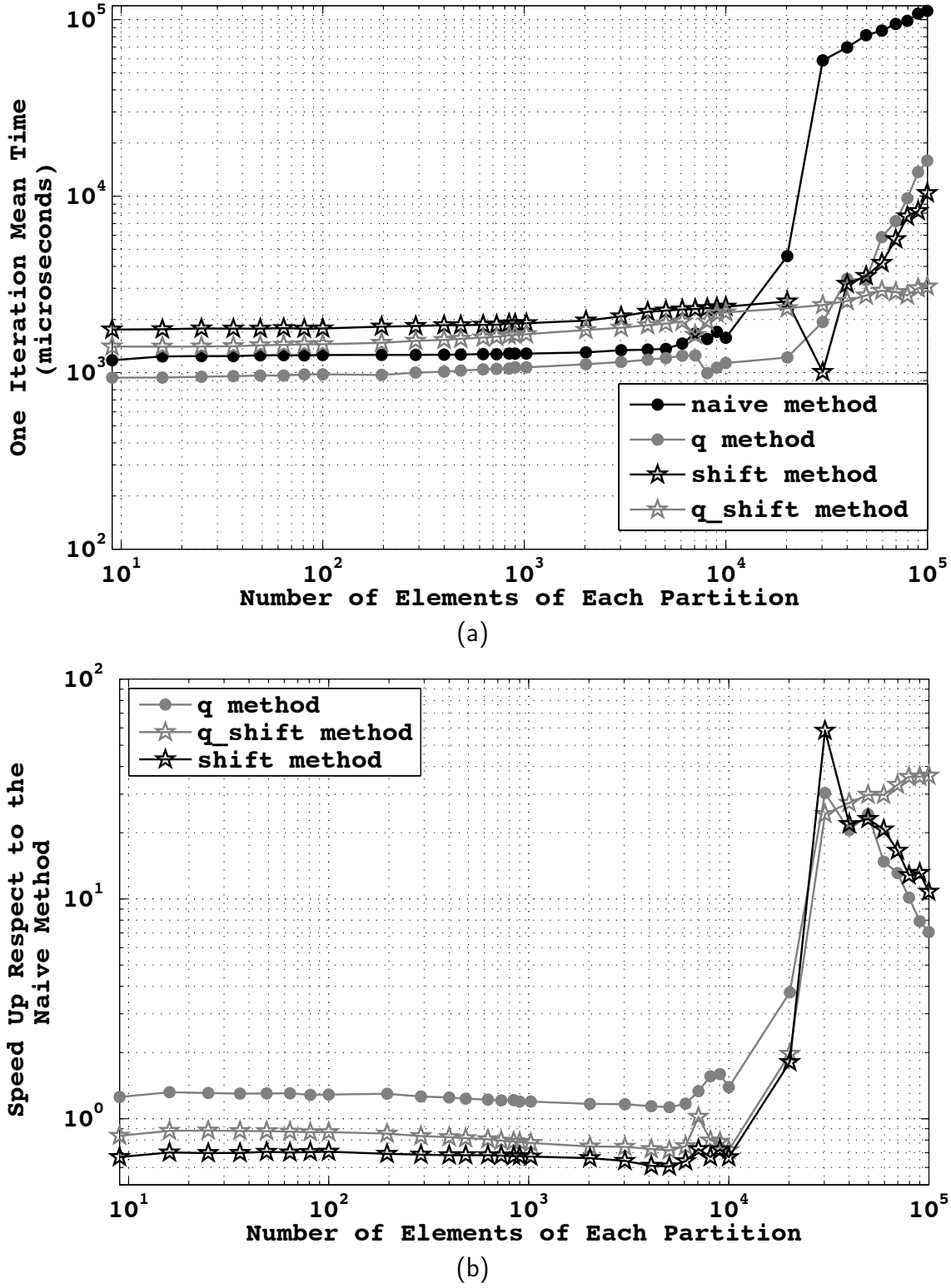
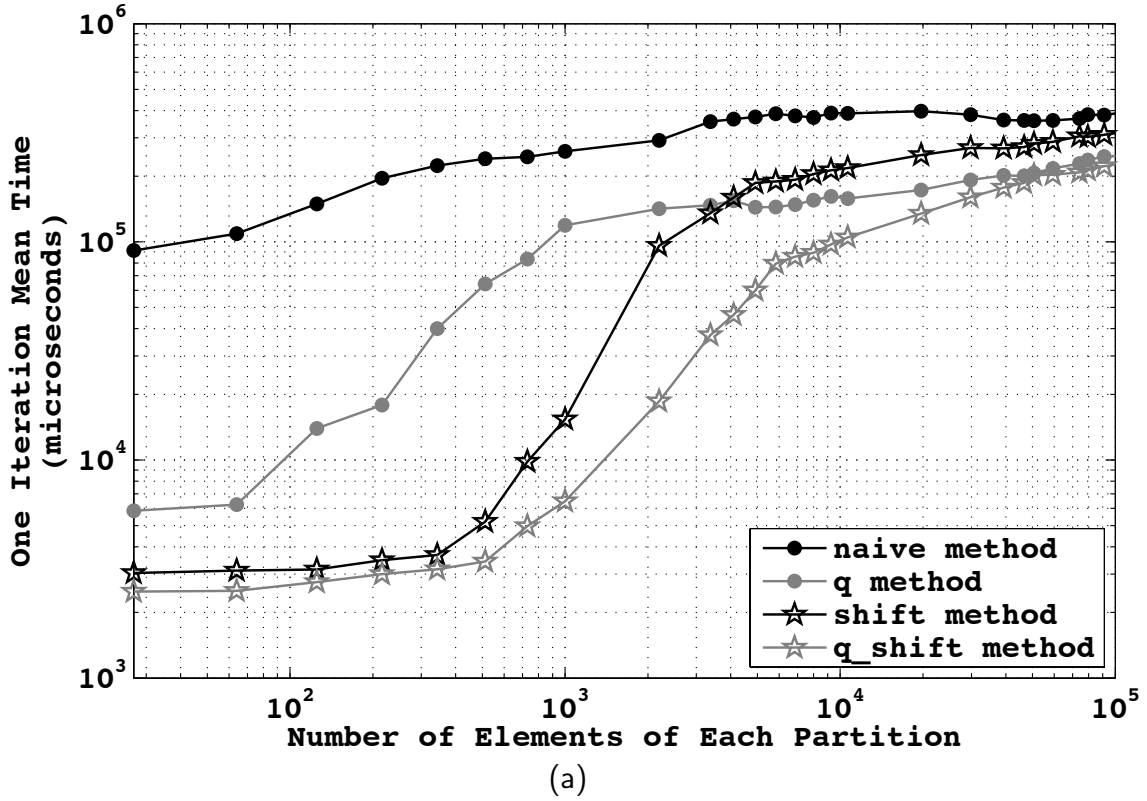
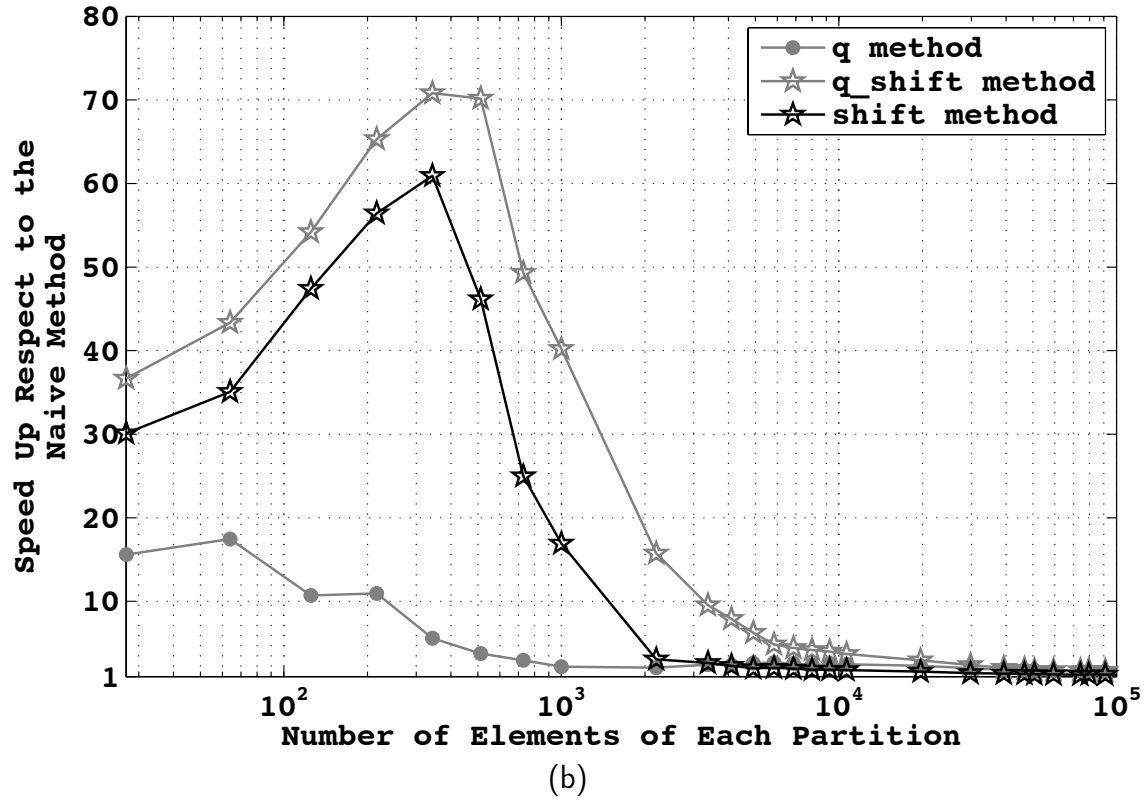


Figure 4.15: Performance results for a nine point stencil in a two-dimensional space, executed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting the MPICH library.



(a)



(b)

Figure 4.16: Performance results for a twenty seven point stencil in a three-dimensional space, executed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting the MPICH library.

naive and $shift$ methods given in the previous Chapter.

The charts in Figure 4.12(b) and Figure 4.13(b) represent the time gain of $shift$, q and q_shift relative to the *naive* method. Irrespective of the number of space dimensions, q_shift offers the best time gain for fine grain parallelization; it is more than four times faster than the *naive* method in the two-dimensional case and nine times faster in the three-dimensional case. Under the same conditions, the $shift$ method offers a time gain of respectively two and five.

The charts in Figure 4.19(a) and Figure 4.19(b) report tests run on the Cell B.E. multi-core. The results, which are presented only for the two-dimensional case, confirm the comments made for the previous architecture. The time gain reported in Figure 4.19(b) is lower than that registered for the Intel architecture. This phenomenon is due to the fact that we are targeting two completely different architectures and also that the $\mathcal{LC}$ support for the Cell is highly optimized and the setup overhead of the communication is reduced to a minimum (see Appendix A).

The charts in Figure 4.15 and Figure 4.16 report communication overheads registered on our Pianosa cluster. In the two-dimensional case, all the methods offer a comparable overhead, but optimizations based on the elimination of diagonal communications, in the $shift$ and q_shift methods, perform worse than the other two.

In the three-dimensional case, the differences between the methods are more relevant. The performance chart in Figure 4.16(b) shows that the q_shift method reaches a time gain of up to seventy relative to the *naive* method while the $shift$ method does not exceed a value of sixty.

4.3.6 Conclusions

Analyzing all the previous performance results on both cluster and multi-core architectures, we can assert that optimizations based on $\mathcal{Q}-transformations$, according to the q or q_shift methods, achieve the lowest communication overhead, irrespective of the number of dimensions in the space and of the target architecture.

4.4 A Closer Analysis of $\mathcal{Q}^{+}-transformations$

The scope of this section is an in-depth investigation of the flow of information exchanged between processes at the concurrent level. In particular we wish to analyse whether the total size of messages sent and received is an invariant between different method implementations.

4.4.1 Analytic Analysis of the Nine Point Stencil

We go back to the cost model of communication which we presented for the $\mathcal{LC}$ language. In the previous Chapter, by exploiting the model, we analyzed the communication overheads of the *naive* and $shift$ methods in the case of the nine point stencil and its multidimensional extensions. We report here the resulting formulae.

$$T_{com}^{naive} = (3^n - 1) * \left(t_{setup}^{snd} + t_{setup}^{rcv}\right) + t_{transm} * s_{total}^{n\_dim\_naive}$$

$$T_{com}^{shift} = (2 * n) * \left(t_{setup}^{snd} + t_{setup}^{rcv}\right) + t_{transm} * s_{total}^{n\_dim\_shift}$$

When we compared the results, we claimed that the two quantities $s_{total}^{n\_dim\_naive}$ $s_{total}^{n\_dim\_shift}$, which represent the sum of the sizes of all sent messages, are equal and that the only difference between the two methods resides in the communication setup components. In the following, we refer to the sum of the sizes of all sent messages as the outgoing information flow.

If we extend the analysis of communication overhead to the q and q_shift methods, we get the following results:

$$T_{com}^{q} = (2^n - 1) * \left(t_{setup}^{snd} + t_{setup}^{rcv}\right) + t_{transm} * s_{total}^{n\_dim\_q}$$

$$T_{com}^{q\_shift} = (n) * \left(t_{setup}^{snd} + t_{setup}^{rcv}\right) + t_{transm} * s_{total}^{n\_dim\_q\_shift}$$

In the case of the nine point stencil, we can claim that the outgoing flow is an invariant for all four methods:

$$s_{total}^{n\_dim\_naive} = s_{total}^{n\_dim\_shift} = s_{total}^{n\_dim\_q} = s_{total}^{n\_dim\_q\_shift}$$

Therefore, the explicit routing of information between processes defined by Plimpton's shift method does not imply any extra data exchange compared to the original stencils. As we see in the next section this is not true for all stencils.

4.4.2 The Jacobi Case

We have looked many times at the Jacobi stencil in the previous Chapter. We briefly analyze it in the light of the previous implementation methods. The shape of the

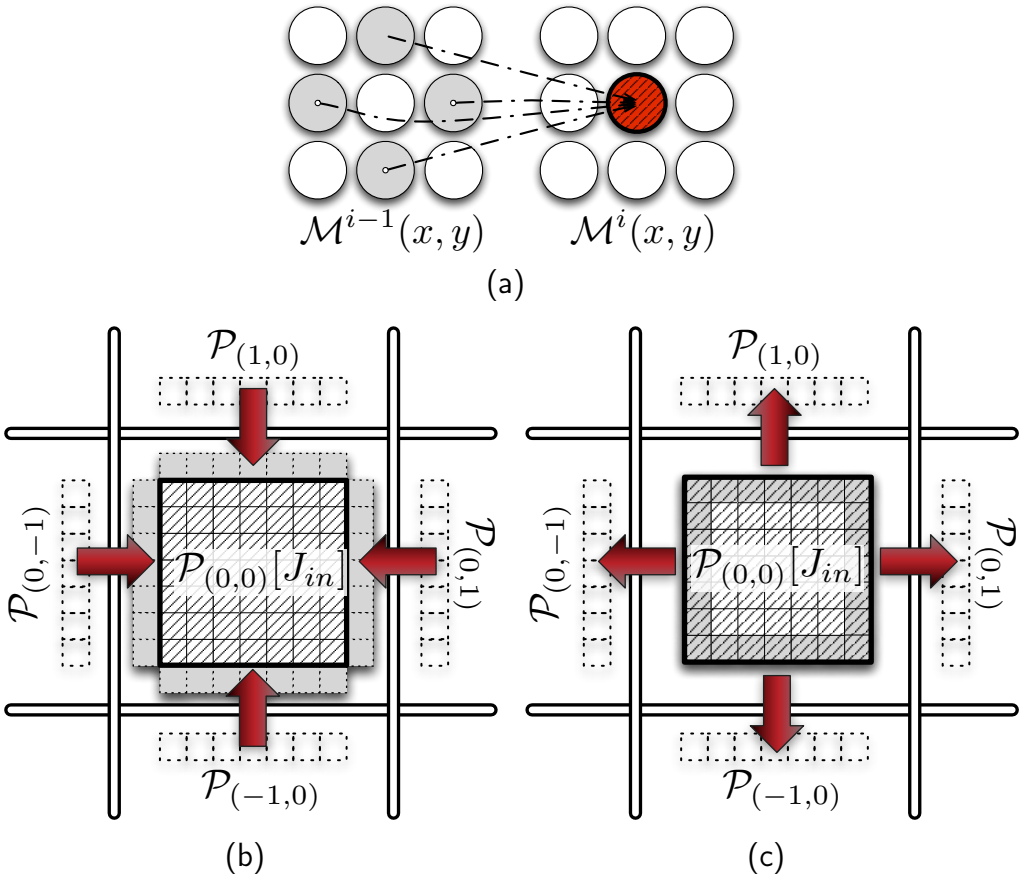


Figure 4.17: Shape 4.17(a) of the Jacobi stencil. Incoming (fig. 4.17(b)) and outgoing (fig. 4.17(c)) communications in a *naive* implementation at the concurrent level.

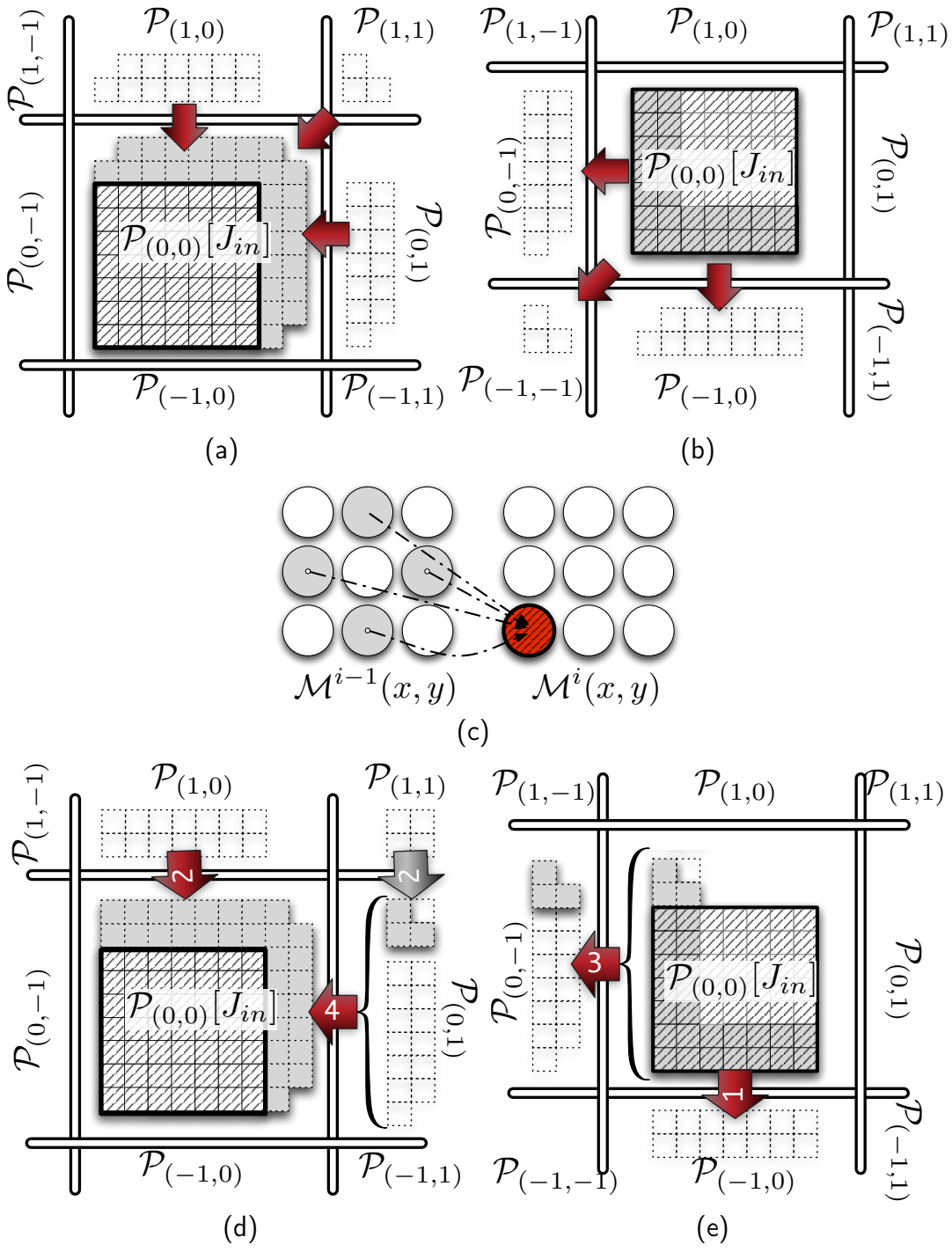


Figure 4.18: Shape 4.18(c) of the $\mathcal{Q}^+[Jacobi]$ stencil. Incoming (fig. 4.18(d)) and outgoing (fig. 4.18(e)) communications in a q implementation at the concurrent level. Incoming (fig. 4.18(a)) and outgoing (fig. 4.18(b)) communications in a $q - shift$ implementation at the concurrent level.

Jacobi stencil is reported in Figure 4.17(a) and a representation of its communication pattern at the concurrent level is shown in Figure 4.17(b) and Figure 4.17(c).

It easy to see that, in contrast to the nine point stencil, the Jacobi stencil does not feature information exchange between neighbours. Hence, the *naive* method is the only one that can be exploited to implement the stencil at the concurrent level. More precisely the $shift$ and *naive* methods result in the same implementation.

As presented in Figure 4.17(b) and Figure 4.17(c), for each step four incoming and four outgoing communications are required. No other optimization can be introduced by the classical approach.

Instead, if we consider the relaxed-safe solution, we can exploit both q and q_shift. This last does not result in the same implementation as q, because for the stencil $\mathcal{Q}^{+}[Jacobi]$, whose relative shape is shown in Figure 4.18(c), a generic partition also features information exchange with some diagonal neighbours.

Exploiting these two methods, both of which come from $\mathcal{Q}-transformations$, we can implement the stencil with respectively only three and two communications as shown in Figure 4.18(a) and Figure 4.18(b) for the q method and in Figure 4.18(a) and Figure 4.18(e) for q_shift.

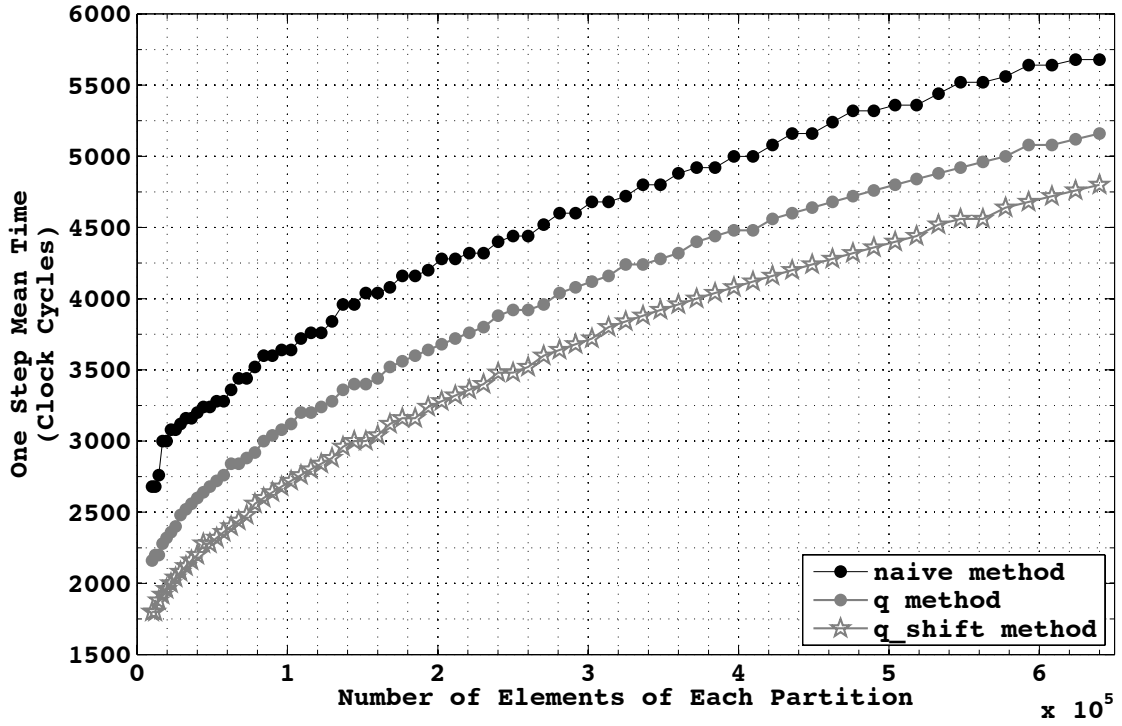
The results that we get for the number of communications of the Jacobi stencil are aligned with the theoretical results we presented and proved. Nevertheless, it is worth pointing out that in the case of the Jacobi stencil we see that the outgoing information flows associated with one step of q and q_shift are higher than for *naive*.

Consider the example of the Jacobi of Figure 4.17 and Figure 4.18, where square partitions are exploited. Suppose that each partition features l elements on each edge. In this configuration, we can assert the following.

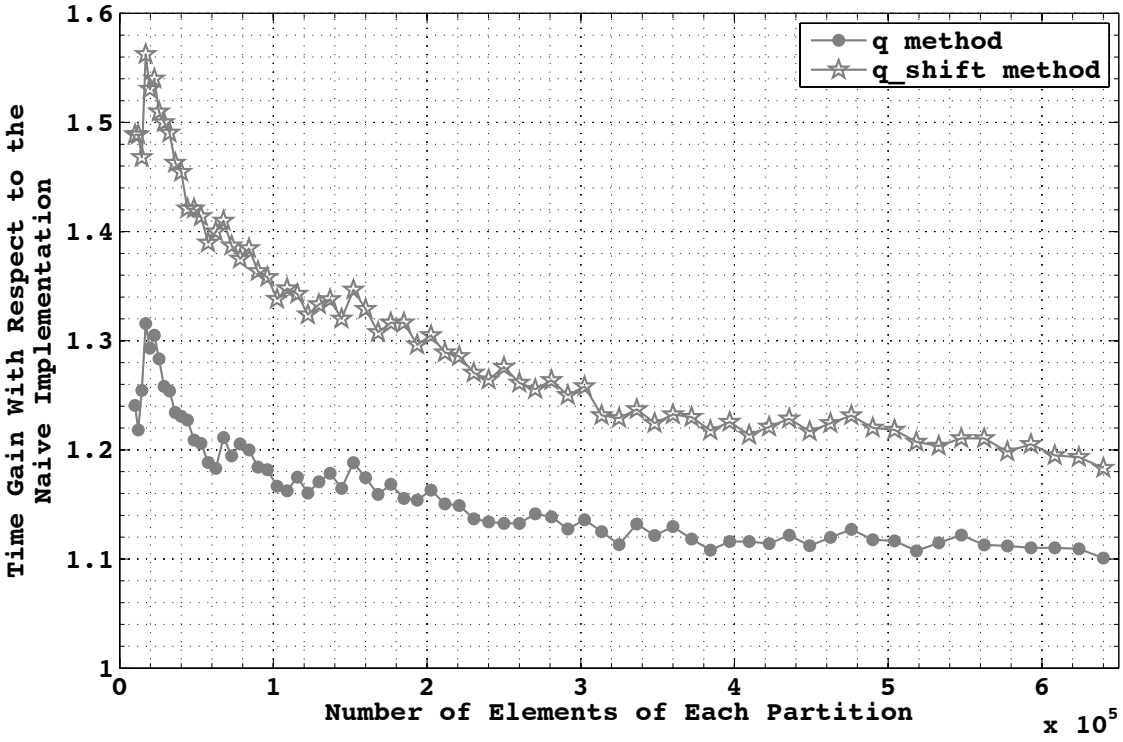
$$\begin{aligned} s_{total}^{n\_dim\_naive} &= 4 * l \\ s_{total}^{n\_dim\_shift} &= 4 * l \\ s_{total}^{n\_dim\_q} &= (4 * l) + 1 \\ s_{total}^{n\_dim\_q\_shift} &= (4 * l) + 2 \end{aligned}$$

The cost of exploiting the q method is to increase the outgoing information flow of one element compared to the *naive* method, while the cost is two elements when exploiting q_shift. Because the increase cannot be greater than the shape volume, it has negligible impact.

In Figure 4.19, we report the charts of performance recorded for the Jacobi example implemented with the three methods. As is evident, the increase of flow has no impact. The optimizations associated with $\mathcal{Q}$–$transformations$ still provide the best performance.



(a)



(b)

Figure 4.19: Jacobi stencil in a two-dimensional space, performed on a Cell B.E. IBM multi-core exploiting the $\mathcal{M}ammu\mathcal{T}$ implementation of $\mathcal{LC}$.

4.5 Negative $\mathcal{Q}-transformations$

In the previous Section, we introduced positive $\mathcal{Q}$–$transformations$. From them, it is possible to define some optimizations, i.e. q and q_shift methods, which reduce the number of communications required to resolve partition dependencies.

These optimizations are based on relaxed-safe transformations, therefore the output of the optimized program is equal to the original one, except for a different organization of the values in the spatial structure (See Definition 2.5.2).

We recall from Equation (4.3) of the proof of Theorem 4.1.1 that we have:

$$\mathcal{M}^i_\psi[e] = \mathcal{M}^i_Q[e - i * q]$$

This is the relation that links the positions of the values in the data structure of the original stencil to the positions in the transformed stencil. In Figure 4.20(a), we show the evolution of the element positions in the working domain step by step for a nine point stencil implemented with the q method. The generic element is translated in each step by a quantity defined by the vector q^+.

It is evident that a complete rearrangement of all the elements is necessary in order to report all the elements in their original positions. Two possible solutions to the problem are the following.

I A sequential post processing computation can rearrange elements in their original positions according to Equation (4.2).

II The features of toroidal space can be exploited to solve the problem. A number of steps, after the end of the computation, can be performed without modifying any value, until the elements are again in the initial positions. This approach implies the execution of a number of *"empty"* steps that depend both on the length of the domain and on the number of steps previously performed by the application.

4.5.1 Defining $\mathcal{Q}^- - transformations$

For the rearrangement problem, a powerful alternative is possible for the class stencils featuring a relative shape with central symmetry relative to the origin. Most of the stencils presented in the literature belong to this class, as do the Jacobi and nine point stencil.

This alternative solution, which requires at most one "empty" step to rearrange the domain elements, consists in the definition of a transformation featuring opposite characteristics to the positive $\mathcal{Q}-transformation$.

Definition 4.5.1 (Negative $\mathcal{Q}-transformation$). Let ψ be a $\mathcal{HUA}$ stencil. A negative $\mathcal{Q}-transformation$ transforms ψ into the $\mathcal{HUA}$ stencil $\mathcal{Q}^-[\psi]$ which is

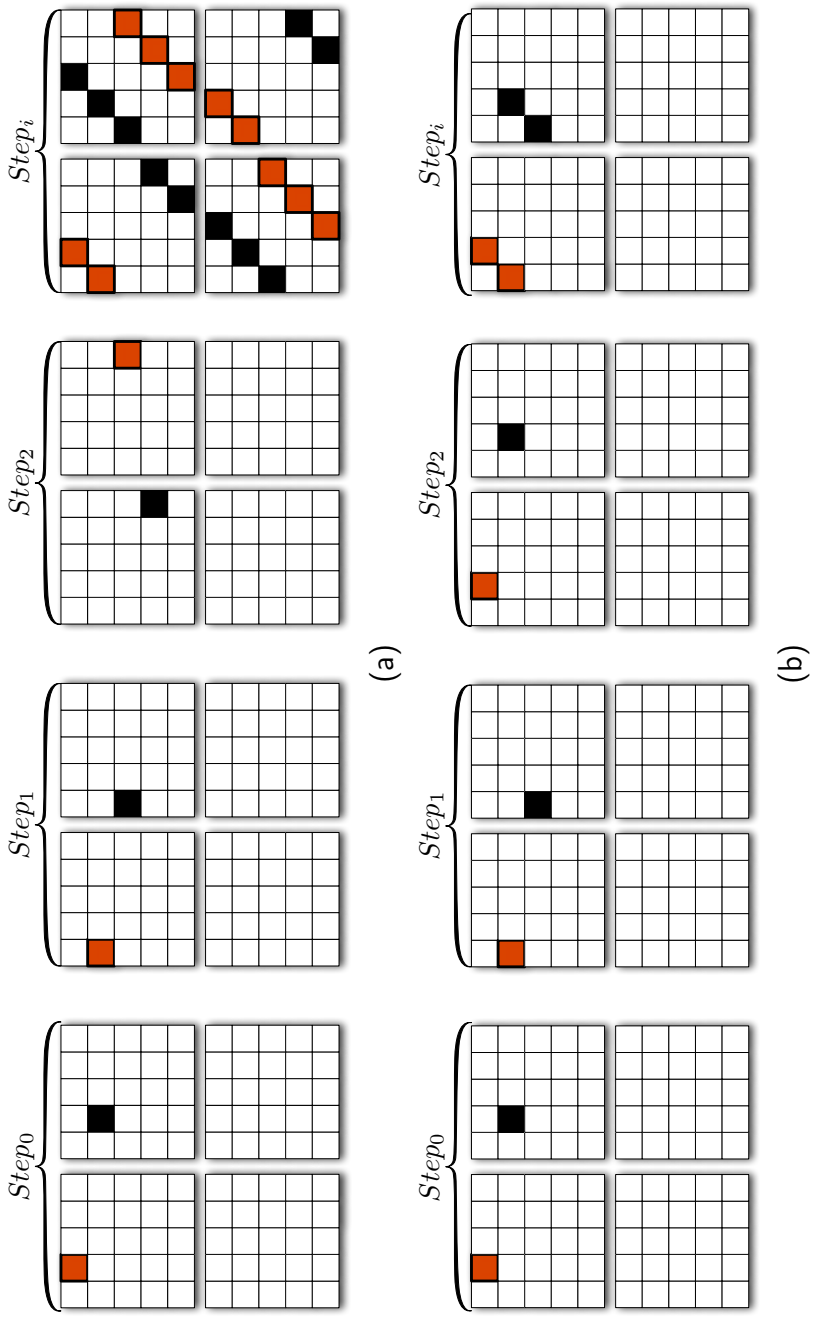


Figure 4.20: Evolution of the element value positions in the spatial structure in the case of a Jacobi stencil defined over a non-toroidal domain space: 4.24(a) with positive $Q-transformation$, 4.24(b) with interleaving of positive and negative $Q-transformation$

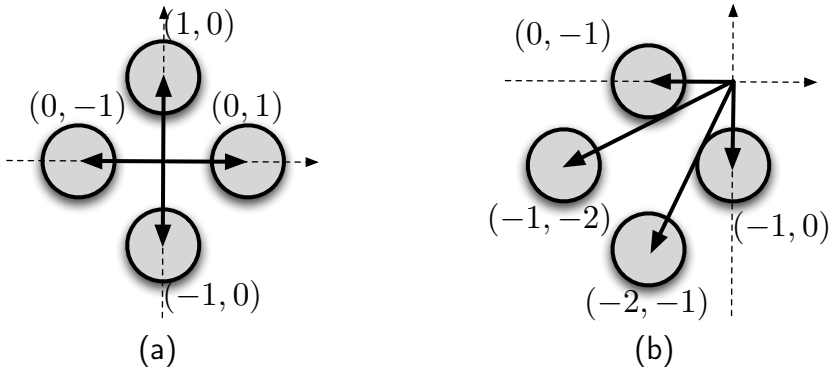


Figure 4.21: Comparison of two graphical representations of the relative shapes of the $Jacobi$ and $\mathcal{Q}^{-}[Jacobi]$ stencils, respectively

```
forall((x, y) ∈ J_in){                                  1
    J_out[x][y] = (J_in[x][y + 1]                       2
    +J_in[x][y − 1] + J_in[x + 1][y]                    3
    +J_in[x − 1][y])/4;                                 4
}                                                       5
```

Figure 4.22: Jacobi pseudo-code.

```
forall((x, y) ∈ J_in){                                  1
    J_out[x][y] = (J_in[x − 1][y]                       2
    +J_in[x − 1][y + 2] + J_in[x][y − 1]                3
    +J_in[x − 2][y − 1])/4;                             4
}                                                       5
```

Figure 4.23: $\mathcal{Q}^{-}[Jacobi]$ pseudo-code.

equivalent to the original one, except for the step model defined as follows:

$$
\begin{aligned}
\forall e \in \mathcal{M} \quad & \overset{\mathcal{Q}^{-}[\psi]}{\rightarrow} \quad \left(\mathcal{F}^{\psi}, \mathcal{S}^{\mathcal{Q}^{-}[\psi]}\right) \\
q^{-} \quad & = \quad < q_1^{-}, \ldots, q_{dim}^{-} > \\
q_i^{-} \quad & = \quad + \max\left\{\beta_\alpha * \epsilon_i \forall \beta_\alpha \in \mathcal{R}\right\} \\
\mathcal{R}^{\mathcal{Q}^{-}[\psi]} \quad & = \quad \mathcal{R}^{\psi} + q \\
\mathcal{S}^{\mathcal{Q}^{-}[\psi]} \quad & = \quad e + \mathcal{R}^{\mathcal{Q}^{-}} \\
& \overset{\Downarrow}{=} \quad e + \left\{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_\alpha = \beta_\alpha + q^{-}\right\} \\
\mathcal{M}^{i+1}_{\mathcal{Q}^{-}[\psi]}[e] \quad & = \quad \mathcal{F}_i(\mathcal{M}^{i}_{\mathcal{Q}^{-}[\psi]}[e + \gamma_1], \ldots, \mathcal{M}^{i}_{\mathcal{Q}^{-}[\psi]}[e + \gamma_n])
\end{aligned}
$$

where $\epsilon = \{\epsilon_1, ...\epsilon_{dim}\}$ is the set of vectors in the natural basis of $\mathbb{N}^{dim}$ and $\beta_\alpha * \epsilon_i$ is the scalar product which returns the component of the vector β_α along the main space direction indicated by the vector ϵ_i. The vector q^{-} is called the **negative vector**.

The key idea is the same as for positive $\mathcal{Q}-transformations$, but the fundamental geometric feature of the negative transformations is that all the components of the γ_α^{-} are now not positive, instead of not negative. Once again we exploit

the Jacobi as a tutorial stencil and we report in Figure 4.21 the representation of both Jacobi and $\mathcal{Q}^{-}[Jacobi]$ relative shapes and in Figure 4.22 and Figure 4.23 the corresponding sequential pseudo-codes.

4.5.2 Combining Positive and Negative $\mathcal{Q}-transformations$

Stencils with centrally symmetric shapes present the following nice characteristic:

$$q^{-} = -q^{+}$$

In other words, a component of q^{-} is the negative of the corresponding component of q^{+}. This feature can be exploited to reduce the effect of element movements, introduced by $\mathcal{Q}-transformations$, as presented in the following Theorem.

Theorem 4.5.1 (**Interleaving Negative and Positive $\mathcal{Q}-transformations$**). *Considering a generic domain space $\mathcal{M}$, let $\mathcal{M}_s^i$ be the state of the domain when applying i times a generic stencil ψ, characterised by central symmetry. Then let $\mathcal{M}_{\mathcal{Q}}^i$ be the state of the domain when applying the stencil $\mathcal{Q}^{+}(\psi)$ on odd steps and the stencil $\mathcal{Q}^{-}(\psi)$ on even ones.*

At the beginning of the odd steps all elements are in their original positions, while at the beginning of the even steps all elements are translated by the quantity q^{-}.

Proof. Let $\mathcal{M}^0$ be the state of the domain before applying any stencil, we have

$$\mathcal{M}^0 = \mathcal{M}_{\psi}^0 = \mathcal{M}_{Q}^0$$

At the first step, we apply the positive $\mathcal{Q}-transformation$: $\mathcal{M}_{Q}^1 = \mathcal{M}_{Q^{+}}^1$. Considering Equation (4.3), which is at the core of the proof of Theorem 4.1.1 on the relaxed safety of general $\mathcal{Q}-transformations$, we obtain the following Equation for positive $\mathcal{Q}-transformations$:

$$\mathcal{M}_{\psi}^1[e] = \mathcal{M}_{Q}^1[e - q^{+}]$$

After the first step, the domain elements are translated relative to the original positions by the quantity q^{-}, indeed, because of the central symmetry of the stencil ψ, we have $-q^{+} = q^{-}$.

At the second step, the negative $\mathcal{Q}-transformation$ is applied and so by Equation (4.3) on page 110 for negative $\mathcal{Q}-transformations$, we get:

$$\mathcal{M}_{\psi}^2[e] = \mathcal{M}_{Q}^2[(e - q^{+}) - q^{-}] = \mathcal{M}_{Q}^2[e + q^{-} - q^{-}] = \mathcal{M}_{Q}^2[e]$$

The domain elements after two steps, with a positive and a negative $\mathcal{Q}-transformation$, are back to their original positions.

Iterating the reasoning, we obtain:

$$\mathcal{M}^i_\psi[e] = \mathcal{M}^i_Q[e + (mod_2(i) * q^-)]$$

At the end of an even number of steps, the data are in their original positions, while after an odd number of steps they are translated by the quantity q^-. □

A graphical presentation of the results derived from Theorem 4.5.1 is given in Figure 4.20(b) for the case of the nine point stencil in a two-dimensional space.

From the previous property, it is clear that, by interleaving positive and negative $\mathcal{Q}-transformations$, the domain does not need any rearranging when the application ends after an even number of steps. In other cases, one "void" step is sufficient.

It is worth highlighting that negative $\mathcal{Q}-transformations$ feature the same characteristics as positive $\mathcal{Q}-transformations$ in terms of impact on partition dependencies. Therefore as for the positive transformations, it is possible to define q and q_shift methods for negative $\mathcal{Q}-transformations$. This means that by interleaving positive and negative $\mathcal{Q}-transformations$, we still achieve an optimization in the number communications in each step.

4.6 Extending $\mathcal{Q}-transformations$ to Semi-Uniform Stencils

The constraint of a toroidal domain space that applies to $\mathcal{HUA}$ stencils makes the translations exploited by $\mathcal{Q}-transformations$ always possible. More precisely, the toroidal space guarantees that, referring to Equation (6.2) on page 195, the generic element $e + \gamma_\alpha$ (which is the element e translated by q) always lies in $\mathcal{M}$.

Concerning $\mathcal{HUA}$ stencil belonging to the semi-uniform class, the entire $\mathcal{Q} - transformation$ formalization can be redefined if we consider the extended version of the stencil. We recall that the existence of this extension is a necessary condition for a stencil to be classified as semi-uniform. Therefore, all the results we have obtained up to now can be directly reused on the extended version because it is a $\mathcal{HUA}$ stencil.

One of the most important results in the extension of $\mathcal{Q}-transformations$ to semi-uniform stencils is their application to stencils which are not defined over a toroidal space; for instance the Jacoibi stencil we presented in Figure 2.17 on page 59.

Figure 4.24 represents the evolution of the values of bound elements, which are highlighted in gray, in the spatial structure in the Jacobi case. The Figure compares both techniques: one that exploits only positive $\mathcal{Q}-transformations$ and another that uses the interleaving method.

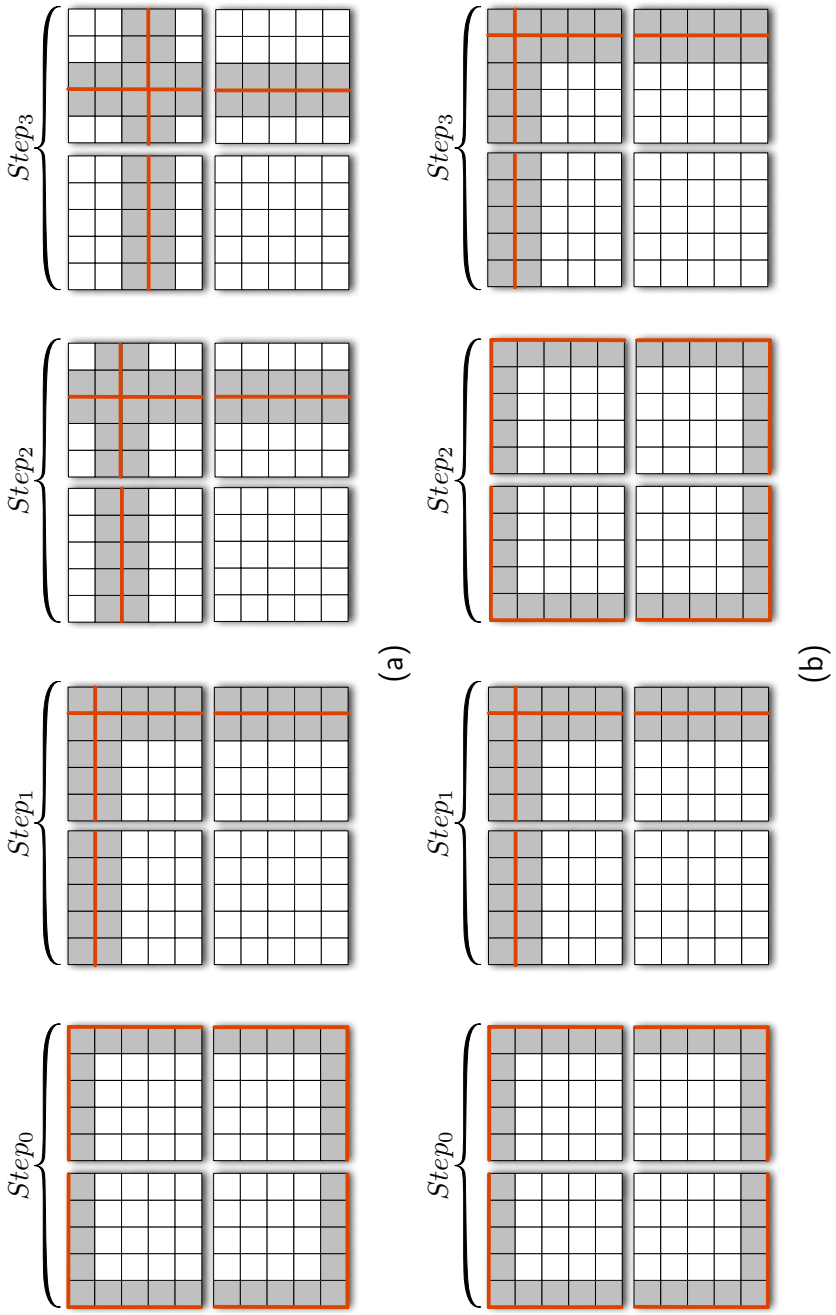


Figure 4.24: Evolution of border elements for the Jacobi stencil defined over a non-toroidal domain space: 4.24(a) with positive $\mathcal{Q}-transformation$, 4.24(b) with interleaving of positive and negative $\mathcal{Q}-transformations$

4.7 Conclusions

In this Chapter, we presented and formally proved the powerful features of $\mathcal{Q}-transformations$: a set of transformations applicable to stencils whose data dependencies can be represented in terms of affine space translations.

In stencil-based parallel applications, communications represent the main overhead, especially when targeting a fine grain parallelization in order to reduce the completion time. Techniques that minimize the number and impact of communications are clearly relevant.

We proved that the reduction of the number of communications featured by $\mathcal{Q}-transformation$-based optimizations is greater than those provided by methods presented in the literature.

Moreover, our experiments both on multi-core and cluster architectures show that implementations exploiting $\mathcal{Q}-transformations$ offer the lowest communication overhead when targeting fine grain parallelizations.

Chapter 5

Step Fusion Transformations

Abstract

The aim of this Chapter is to introduce a new and powerful class of stencil transformations called $\mathcal{Q}-\mathcal{S}tep-\mathcal{F}usion$ ($\mathcal{QSF}$).

Data parallelism is based on data distribution and function replication. A well-known technique, called *ghost cell expansion*, aims to reduce communication overheads by exploiting data replication. The transformation, which in the literature is expressed as a transformation at the concurrent level, introduces an interesting trade-off between a reduction in communication overhead and an increase in computational load.

The $\mathcal{QSF}-transformations$ expand and restructure the main concept of the *ghost cell expansion* into a definition of a new class of stencil transformations which are defined at the functional dependency level instead of the concurrent one. This new in-depth point of view of the *ghost cell expansion* technique, makes possible the exploitation of both results of $\mathcal{Q}-transformations$, for communication overhead reduction, and of new optimizations which are focused on lowering the computation load.

In this Chapter, we demonstrate analytically the benefit of the new transformations and we validate the results with a complete set of experiments on different kinds of computational architecture.

The Chapter is structured as follows. Section 5.1 introduces the *oversending* technique and explains how data replication is exploited to reduce the communication overhead.

Section 5.2 introduces $\mathcal{SF}-transformations$, which come from a revisit of oversending as a stencil transformation defined at the functional dependency level.

Finally, Section 5.3 analyses the performance improvements of $\mathcal{SF}-transformations$ on sequential computational kernels which come from the effects of temporal locality.

Contents

5.1 Data Replication in $\mathcal{HUA}$ Stencils

The data parallel paradigm is based on data partitioning and replication of functions. In terms of the $\mathcal{HUA}$ model, the working domain is distributed amongst multiple processes and the computational kernels, represented by step functions, are replicated.

In the previous Chapters, we have seen the overheads that in data parallel programs are introduced by communications between processes. We have presented a set of optimizations (some already cited in the literature but also others that are new such as the $\mathcal{Q}-transformations$) which target the reduction of these overheads.

A common characteristic of all the optimization methods is the exploitation of strategies that pack smaller incoming or outgoing messages together. The benefit comes from the reduction of the setup time component of the communication overheads.

Ding and He in [16] propose a technique that we call *oversending*, but is also known as *Ghost Cell Expansion*. The technique, which **is expressed at the concurrent level**, exploits data replications to reduce the mean number of communications per step.

In Section 5.1.1 and Section 5.1.2, we present the oversending method and we analyze its drawbacks. Moreover we compare the new technique with the $\mathcal{Q}-transformations$ and we present a fusion of the two transformations.

5.1.1 The Oversending Method

We start the explanation of the oversending method by looking at the Jacobi stencil example in a two-dimensional space. For the sake of simplicity, we break a clause of Working Hypothesis 3.1 and consider row partitioning.

In row partitioning, the *naive* and $shift$ methods are the same, therefore in the rest of the Section we can consider only the *naive* method. Nevertheless all the following issues can be easily extended to block partitioning independently from the fact that a *naive* or a $shift$ method has been targeted.

The pseudo-code of a program that implements the Jacobi stencil at the concurrent level by exploiting a row partition strategy is reported in Figure 5.1. As previously done, to avoid making the pseudo-code notation heavier, we leave to Figure 5.5 on page 154 the burden of graphically highlighting the elements that are involved in the communications.

The oversending method is based on the principle of sending and symmetrically receiving more data during some steps in such a way that communications between processes are not required at each stencil step.

In a generic step i, a process exchanges longer messages than the *naive* implementation. The messages are composed of two sets λ and δ of values of elements in $\mathcal{M}^i$.

```
int c = 512;                                                    1
int r = 50;                                                     2
double J_in[r][c], J_out[r][c];                                 3
load_partition_values(J_in);                                    4
                                                                5
for(i_step = 0; i_step < 10; i_step++){                         6
                                                                7
 SEND(P_(-1)); SEND(P_(+1));                                    8
                                                                9
 COMPUTE(Incoming independent region);                          10
                                                               11
 RECV(P_(-1)); RECV(P_(+1));                                   12
                                                               13
 COMPUTE(Incoming dependent region);                           14
                                                               15
 swap(J_in, J_out);                                            16
}                                                              17
return_partition(J_out);                                       18
```

Figure 5.1: Representation in pseudo-code of a Jacobi stencil implemented at the concurrent level, exploiting the naive method and a row partitioning strategy.

I The δ set represents the same information that is exchanged via messages defined in the *naive* method. The values of δ are required to compute the values of the incoming dependent region at time $i + 1$.

II The set λ is used to update the values of the δ elements at time $i + 1$ too. Indeed, the new updated values of δ can resolve the functional dependencies of the incoming dependent region also for step $i + 1$. In this way, it is possible to compute the values at time $i + 2$ of all the partition elements.

All the data received during one step resolve the partition dependencies for two steps. This important result does not come for free. The elements of δ have to be computed, or better updated, before they can be exploited in the step featuring no communications. The update computation represents the additional computational overhead compared to the naive implementation.

The elements in δ represent replicated data. Both the sender and receiver of these data are going to independently update them according to the computational kernel defined by the stencil step function. In other words, the same set of values are computed in the same step by more than one process.

The pseudo-code of a Jacobi stencil implementation that has been optimized by the oversending method is reported in Figure 5.2. Once again, we leave to Figure 5.5

on page 154 the burden of graphically highlighting the elements that are involved in the communications.

Oversending, as it is clear from the method description in pseudo-code, increases the amount of data sent in one step in order to replace the communications in the following step with some computations. The method can be extended in order to avoid communications in more than one consecutive step. Hence, we say that the oversending method features different levels of optimizations.

Unlike previous methods, the levels give to the method a parametric feature: it is possible to decide how long is the sequence of steps that do not perform communications. Nevertheless, the higher the level of optimization is (i.e. the more steps that avoid communications), the higher the extra computational load. This relation creates an interesting trade-off between communication and computation.

5.1.2 Oversending Performance Model

We now focus on performance studies of the oversending method. We analyze computation and communication times. For the sake of simplicity, we do not take into consideration the possible overlapping between the two components.

In Figure 5.3, we show a graphical representation of the evolution of data structures and communications in the oversending program of Figure 5.2. The representation can be used to easily understand how the performance model of the program is extracted.

Starting with the computation component, we see that in two steps, of which only one features communications, a process computes two times the elements of its partition and moreover it updates the elements of two received rows.

Consider t_{one_el} the time to update a single element according to the Jacobi stencil and moreover let the variables r and c be respectively the number of rows per partition and the number of elements in a row. We can model the mean computation time for a single Jacobi step as follows:

$$T_{comp} = t_{one\_el} * c * (r + 1)$$

With regard to the communication component, each process features two send and two receive operations, whose associated messages consist of two rows of elements. The send and receive operations are computed in only one step out of two. Therefore, referring to the size of a row with the term $s(R)$, we can model the mean communication overheads for a single Jacobi step as follows:

$$T_{com} = (t_{set}^{snd} + t_{set}^{rcv}) + 2 * t_{trasm} * s(R)$$

We extend the model of the Jacobi example to a generic level $\mathcal{L}$ of oversending. In our convention, we consider $\mathcal{L} = j$ the level of optimization where only one step out of j features communications.

```
int c = 512;                                                          1
int r = 50;                                                           2
double J_in[r][c], J_out[r][c];                                       3
load_partition_values(J_in);                                          4
                                                                      5
for(i_step = 0; i_step < 10; i_step++){                               6
                                                                      7
    if(i_step%2 = 0){                                                 8
        SEND(P_(0)); SEND(P_(+1));                                    9
                                                                      10
        COMPUTE(Incoming independent region);                         11
                                                                      12
        RECV(P_(0)); RECV(P_(+1));                                    13
                                                                      14
        COMPUTE(Incoming dependent region);                           15
                                                                      16
    }else{                                                            17
        SEND(P_(0)); SEND(P_(+1));                                    18
                                                                      19
        COMPUTE(Incoming independent region);                         20
                                                                      21
        UPDATE_REPLICATED_DATA();                                     22
                                                                      23
        COMPUTE(Incoming dependent region);                           24
    }                                                                 25
    swap(J_in, J_out);                                                26
}                                                                     27
return_partition(J_out);                                              28
```

Figure 5.2: Representation in pseudo-code of a Jacobi stencil implemented at concurrent level by an oversending method and a row partitioning strategy.

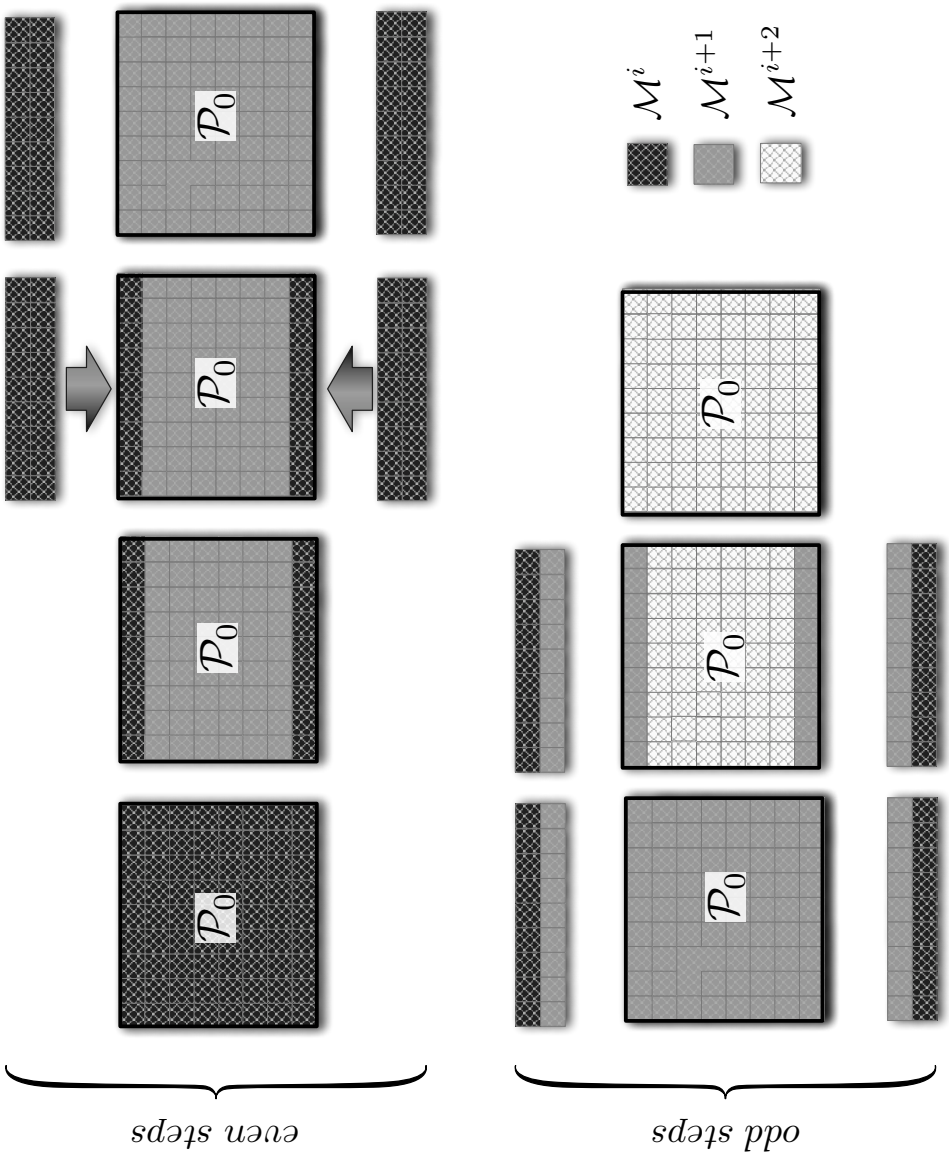


Figure 5.3: Representation of the evolution of the data structure elements and communications during the computation of a Jacobi stencil that has been optimized by the oversending method according to the pseudo-code in Figure 5.2.

Figure 5.4: Representation of the evolution of the data structure elements and communications during the computation of a Jacobi stencil that has been optimized by the q-oversending method.

$$T_{comp} = t_{one\_el} * c * (r + \mathcal{L} - 1)$$

$$T_{com} = \frac{(t_{set}^{snd} + t_{set}^{rcv})}{\mathcal{L} - 1} + 2 * t_{trasm} * s(R)$$

From the previous formula, the following two main aspects are apparent.

I As with all previous methods for communication optimization, the oversending method targets the reduction of the setup overhead of communications. The mean flow, i.e. the amount of data sent or received, per step does not decrease.

II The reduction of communication overheads comes from an increase in computational load: more elements have to be updated. Therefore, for each configuration there is an optimum value of $\mathcal{L}$. The value depends on the performance of the physical architecture and on the computational characteristics associated with the target stencil.

5.1.3 Oversending and $\mathcal{Q}-transformations$

In Table 5.1, we compare performance results for the oversending method with results for the *naive* and q methods in the case of the Jacobi example. Figure 5.5 shows the communication pattern for each of the compared methods.

We recall that because we are considering a row partitioning strategy in the interest of simplicity, the implementations of the q and $q-shift$ methods are equivalent.

By carefully observing the data reported in Table 5.1, it becomes apparent that the performance of the q method recovers the advantages of the other two methods. More formally we can write the two following equations:

$$T_{com}^{q} = \min\left\{T_{com}^{oversending}, T_{com}^{naive}\right\}$$

$$T_{comp}^{q} = \min\left\{T_{comp}^{oversending}, T_{comp}^{naive}\right\}$$

Therefore we can claim that in the case of row partitioning, the q method achieves the same performance improvement as the oversending method without increasing the computation time. We can extend the previous observation with the following Theorem.

Theorem 5.1.1 (**Comparison of Oversending and $\mathcal{Q}-transformations$**). *In the case of block partitioning and with a stencil ψ that features partition dependencies with all its neighbours, the $q-shift$ method achieves the same reduction of communication overheads as the oversending method without introducing the extra computational load.*

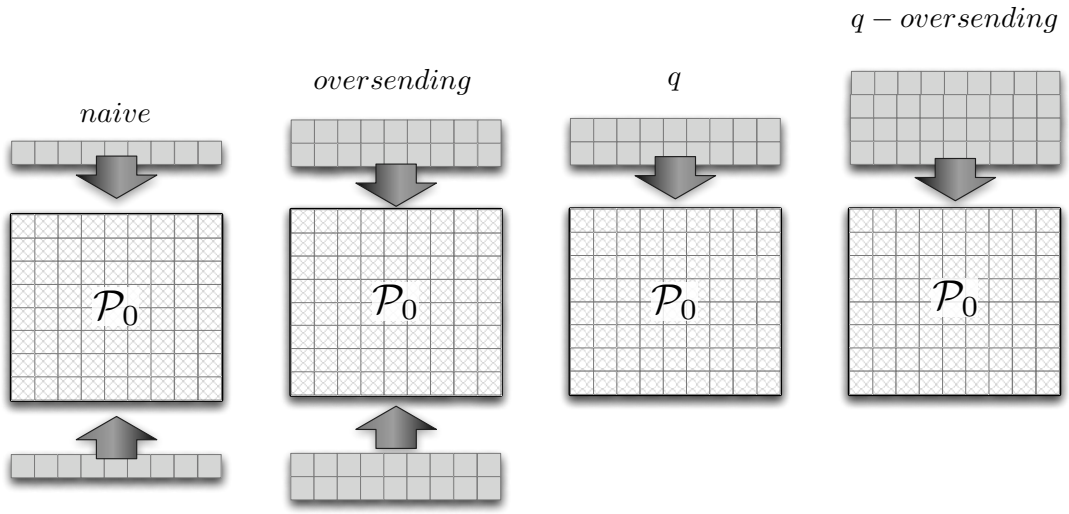


Figure 5.5: Graphical representation of the communication patterns of *naive*, q, *oversending* and $q-oversending$ methods used to implement a Jacobi stencil in a row partition configuration.

Jacobi	T_{com}	T_{comp}
$Naive$	$\left(2 * (t_{set}^{snd} + t_{set}^{rcv}) + 2 * t_{trasm} * s(R)\right)$	$\left(t_{one\_el} * c * r\right)$
$Oversending_{(\mathcal{L}=2)}$	$\left((t_{set}^{snd} + t_{set}^{rcv}) + 2 * t_{trasm} * s(R)\right)$	$\left(t_{one\_el} * c * (r+1)\right)$
q	$\left((t_{set}^{snd} + t_{set}^{rcv}) + 2 * t_{trasm} * s(R)\right)$	$\left(t_{one\_el} * c * r\right)$
$q - oversending_{(\mathcal{L}=2)}$	$\left(\frac{(t_{set}^{snd} + t_{set}^{rcv})}{2} + 2 * t_{trasm} * s(R)\right)$	$\left(t_{one\_el} * c * (r+1)\right)$

Table 5.1: Performance model of the communications and computations in *naive*, q, *oversending* and $q-oversending$ methods used to implement a Jacobi stencil in a row partition configuration. The parameter considered is mean time per step.

Proof. We know that the $q - shift$ method implies, for a stencil ψ defined over an n-dimensional space and featuring partition dependencies with all its neighbours, a number of communications equal to n.

The oversending method, with an optimization level $\mathcal{L}$ equal to two, requires communications in one step out of two. If in addition to the oversending method, we attempt to use the shift method to reduce the number of diagonal communications, the resulting implementation requires a number of communications equal to $2 * n$ in every two steps. Therefore, on average, it requires n communications per step.

Therefore, the two solutions feature the same number of communications per step. Nevertheless the solutions associated with the oversending methods also introduce extra computational overhead in order to update the replicated data. □

As we stressed at the beginning of the Section, the oversending method is an optimization that is defined in the literature at the concurrent level. We can therefore mix $\mathcal{Q} - transformations$, which are defined at the functional dependency level, with the oversending method without any problem. We call the resulting method $q - oversending$.

In the case of the Jacobi stencil, a graphical representation of the communication pattern and communications implied by the $q - oversending$ method is reported in Figure 5.4 and its performance model is reported on the last row of Table 5.1.

As a result, the new method halves the communication setup compared to the other two methods and increases the computation load to that of the oversending method. Thanks to the multiple level of optimization, The $q - oversending$ method enhances both $q - shift$ and q with a parametric feature that neither has.

5.2 Step Fusion Transformations

In the previous Sections, we have introduced and analyzed the oversending method as defined in the literature: an optimization for communications at the concurrent level. In this Section, we focus on the definition of a new model of oversending. Indeed, we are going to redefine the method as a stencil transformation at the functional dependency level. As we will see later in the Chapter, this solution provides sequential computation kernels for new optimizations which better exploit the cache hierarchy with a higher temporal locality.

5.2.1 A Structured Interpretation of Oversending

We introduce the $\mathcal{S}tep-\mathcal{F}usion$ transformations ($\mathcal{SF}$), which are a structured redefinition of the *oversending* technique. The main concept of the new transformations is to merge one or more steps of a $\mathcal{HUA}$ stencil into a single one.

Consider the Equation that is at the core of the classic definition of the *Jacobi* stencil:

$$\begin{aligned}\mathcal{M}^{i}[x][y] &= \Big(\mathcal{M}^{i-1}[(x,y+1)] + \mathcal{M}^{i-1}[(x,y-1)] + \\ &\quad +\mathcal{M}^{i-1}[(x+1,y)] + \mathcal{M}^{i-1}[(x-1,y)]\Big)/4 \end{aligned} \tag{5.1}$$

The formula claims that the computation of an element value at time i depends on the evaluation of some elements at time $i-1$. In turn, the requested values can be indirectly estimated by applying the definition of the *Jacobi* stencil and using evaluations of elements at time $i-2$.

For example, the value $\mathcal{M}^{i-1}[(x,y+1)]$, which is a component of Equation (5.1), can be expressed as:

$$\begin{aligned}\mathcal{M}^{i-1}[(x,y+1)] &= \Big(\mathcal{M}^{i-2}[(x+1,y+1)] + \mathcal{M}^{i-2}[(x-1,y+1)] \\ &\quad +\mathcal{M}^{i-2}[(x,y+2)] + \mathcal{M}^{i-2}[(x,y)]\Big)/4\end{aligned}$$

We can therefore redefine the *Jacobi* Equation (5.1) as a function of values of the domain elements at time $i-2$ instead of $i-1$.

$$\begin{aligned}\mathcal{M}^{k}[(x,y)] &= \Big(4 * \mathcal{M}^{k-2}[(x,y)] + 2 * \mathcal{M}^{k-2}[(y+1,y+1)] + \\ &\quad +2 * \mathcal{M}^{k-2}[(x-1,y-1)] + 2 * \mathcal{M}^{k-2}[(x-1,y+1)] + \\ &\quad +2 * \mathcal{M}^{k-2}[(x+1,y-1)] + \mathcal{M}^{k-2}[(x+2,y)] + \\ &\quad +\mathcal{M}^{k-2}[(x-2,y)] + \mathcal{M}^{k-2}[(x,y+2)] + \\ &\quad \mathcal{M}^{k-2}[(x,y-2)]\Big)/16\end{aligned} \tag{5.2}$$

The new *Jacobi* Equation (5.2) can be interpreted as a new alternative stencil which fuses two steps of the original *Jacobi* into one.

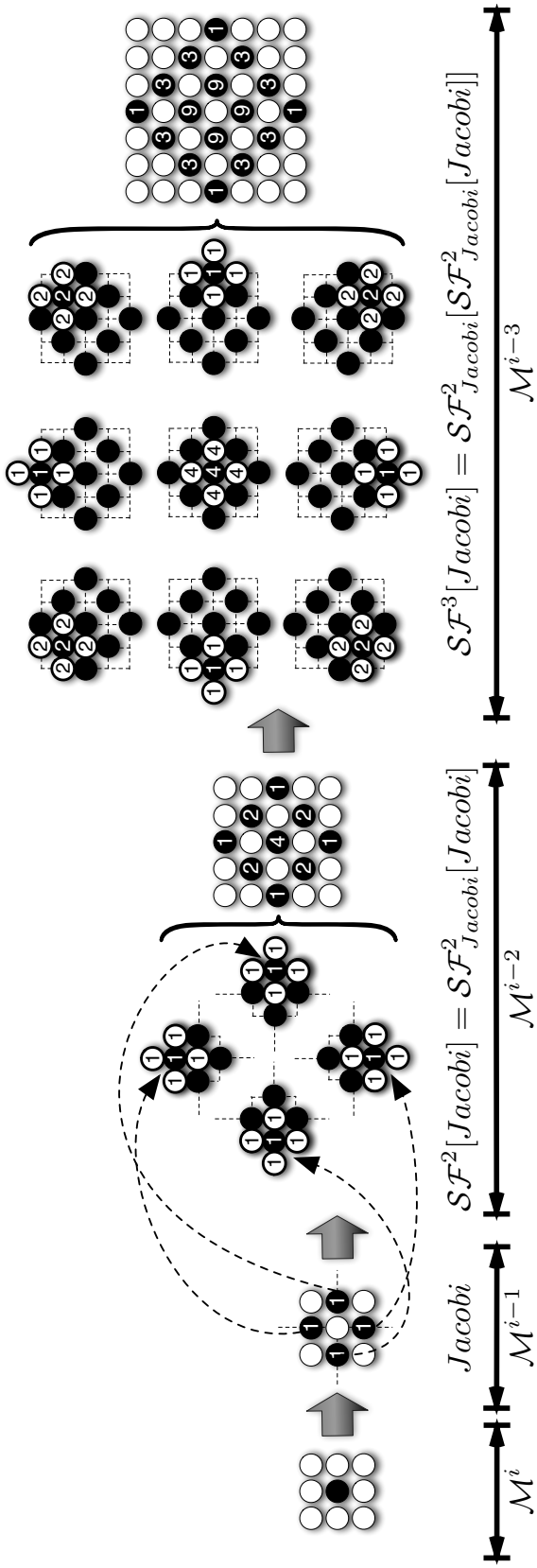


Figure 5.6: Graphical representation of the step fusion transformations applied to a Jacobi stencil.

The procedure can be iterated more times, therefore it is possible to define a generic level of fusion. Merging k steps of a stencil we can compute the values of the working domain elements from a knowledge of their values at time $i-k$ as follows.

$$\mathcal{M}^i[x][y] = \mathcal{F}\Big(\mathcal{M}^{i-k}[..][..] \dots \mathcal{M}^{i-k}[..][..]\Big)$$

We model this new stencil, that comes from the fusion of one or more steps of the original one, as the result of a transformation that is defined at the functional dependency level: $\mathcal{SF}-transformation$.

5.2.2 Formal Definition of $\mathcal{SF}$ Transformations and Their Properties

Definition 5.2.1 (**$\mathcal{S}$tep $\mathcal{F}$usion transformations).** The $\mathcal{SF}^k$ is a $\mathcal{S}$tep $\mathcal{F}$usion of k levels, where a generic stencil ψ (expressed in the $\mathcal{HUA}$ model) is transformed into the stencil $\mathcal{SF}^k[\psi]$, defined recursively as:

$$\begin{cases} \mathcal{SF}^k[\psi] = & \mathcal{SF}^2_\psi\Big[\mathcal{SF}^{k-1}[\psi]\Big] \\ \mathcal{SF}^2[\psi] = & \mathcal{SF}^2_\psi\big[\psi\big] \\ \mathcal{SF}^1[\psi] = & \psi \end{cases} \tag{5.3}$$

$\mathcal{SF}^2_\psi$ is a specific support transformation which depends on a defined stencil, in this case ψ, and maps $\mathcal{HUA}$ stencils onto $\mathcal{HUA}$ stencils.

Definition 5.2.2 (**$\mathcal{S}$tep $\mathcal{F}$usion Support Transformations).** Let ψ and χ be two $\mathcal{HUA}$ stencils which are characterized respectively by the couples $(\mathcal{F}^\psi, \mathcal{S}^\psi)$ and $(\mathcal{F}^\chi, \mathcal{S}^\chi)$ and let $\mathcal{S}^\chi(e) = \{g_1, \dots, g_n\}$ and $\mathcal{S}^\psi(e) = \{\omega^e_1, \dots, \omega^e_m\}$. The transformed stencil $\mathcal{SF}^2_\psi[\chi]$ is a $\mathcal{HUA}$ stencil characterized by the following step model.

$$\begin{aligned} \forall e \in \mathcal{M} \quad & \overset{\mathcal{SF}^2_\psi[\chi_{step_i}]}{\rightarrow} \quad \big(\mathcal{F}^\chi{}_i, \mathcal{F}^\psi{}_{i-1}, \mathcal{SF}^2_\psi[\mathcal{S}^\chi_i]\big) \\ \mathcal{SF}^2_\psi[\mathcal{S}^\chi{}_i] \quad & = \quad \left\{ \forall \omega \in \mathcal{M} | \omega \in \bigcup_{g_\alpha \in \mathcal{S}^\chi_i(e)} \mathcal{S}^\psi_{i-1}(g_\alpha), \right\} \\ \mathcal{M}^{i+1}[e] \quad & = \quad \mathcal{F}^\chi_i\Big(\mathcal{F}^\psi_{i-1}(\mathcal{M}^{i-1}[\omega^{g_1}_1], \dots, \mathcal{M}^{i-1}[\omega^{g_1}_m]), \dots, \\ & \qquad \dots, \mathcal{F}^\psi_{i-1}(\mathcal{M}^{i-1}[\omega^{g_n}_1], \dots, \mathcal{M}^{i-1}[\omega^{g_n}_m])\Big) \\ & \quad where \quad \{\omega^{g_\alpha}_1 \dots \omega^{g_\alpha}_m\} = \mathcal{S}^\psi_{i-1}(g_\alpha) \end{aligned} \tag{5.4}$$

By the definition of a $\mathcal{HUA}$ stencil, we know that at step $i+1$ χ updates an element e of the working domain with the result of the computation of the step

function $\mathcal{F}^{\chi}$ with the evaluations of the $\mathcal{S}^{\chi}(e)$ set elements as parameters. In this scenario, $\mathcal{SF}^2_{\psi}[\chi]$, a support transformation for the ψ stencil applied to χ, defines the shape set $\mathcal{SF}^2_{\psi}[\mathcal{S}^{\chi}{}_i]$ replacing each element g_α in $\mathcal{S}^{\chi}(e)$ with the domain element set $\mathcal{S}^{\chi}(g_\alpha)$. $\mathcal{S}^{\chi}(g_\alpha)$ is the domain element set which defines the shape of ψ featuring g_α as its application point. The resulting set $\mathcal{SF}^2_{\psi}[\mathcal{S}^{\chi}{}_i]$ constitutes the shape of the transformed stencil.

Figure 5.6 presents graphically the previous concept, describing how support transformations work when $\mathcal{QSF}-transformations$ are applied to a *Jacobi* stencil.

In the case of a two level step fusion, the stencil that we obtain is $\mathcal{SF}^2[Jacobi] = \mathcal{SF}^2_{Jacobi}\big[Jacobi\big]$, which features nine elements, five more than the original. A single step of $\mathcal{SF}^2[Jacobi]$ corresponds to two iterations of the original *Jacobi*. Considering the time model of the original Jacobi stencil, we see that, in order to produce element values at time $i+1$, $\mathcal{SF}^2[Jacobi]$ requires values at time $i-1$.

When applying a $\mathcal{SF}$ with three levels of step fusion, we get the sixteen element stencil $\mathcal{SF}^3\big[Jacobi\big]$, whose single computation step is equivalent to three applications of the *Jacobi* step. Therefore, to produce values at step i, the transformed stencil requires domain values at step $i-2$.

From Figure 5.6, it is also possible to appreciate the role of the support transformation. We review the formal definition of support function for three levels of step fusion applied to *Jacobi*. By definition, we have:

$$\mathcal{SF}^3\big[Jacobi\big] = \mathcal{SF}^2_{Jacobi}\Big[\mathcal{SF}^2_{Jacobi}\big[Jacobi\big]\Big]$$

Therefore, according to the notations used in the definition, ψ and χ are respectively *Jacobi* and $\mathcal{SF}^2_{Jacobi}\big[Jacobi\big]$, while $\mathcal{S}^{\psi}{}_i$ and $\mathcal{S}^{\chi}{}_i$ are the two stencil shapes.

The procedure to obtain $\mathcal{SF}^2_{\psi}[\mathcal{S}^{\chi}{}_i]$ consists in replacing each element in the χ shape ($\mathcal{S}^{\chi}{}_i$) with the elements of the *Jacobi* shape ($\mathcal{S}^{\psi}{}_i$) featuring the replaced element as application point. The union of all the new elements gives as result the shape of the sixteen element stencil defined by $\mathcal{SF}^3\big[Jacobi\big]$.

In Figure 5.7 and Figure 5.8 we also report the graphical representation of $\mathcal{SF}-transformations$ for the Laplace and nine point stencils. It is evident that the higher the level of fusion, the wider the stencil shape becomes.

In all the Figures representing stencils which are the results of $\mathcal{SF}$–$transformations$, a parameter is associated to each shape element. The parameter is the integer part of the coefficient used by the step function. To better understand the previous explanation, it is sufficient to compare the coefficients of $\mathcal{SF}^2[Jacobi]$ in Figure 5.6 and Equation (5.2). We just consider the integer part for the sake of simplicity in the graphical representations.

In the following, we focus on the definition and proof of two significant properties of $\mathcal{QSF}-transformations$.

Theorem 5.2.1 (**$\mathcal{SF}$s Correctness**). *Let ψ be a generic stencil and let $\mathcal{SF}^k[\psi]$ be its transformation with respect to $\mathcal{SF}-transformations$. The values of a domain $\mathcal{M}$ after k steps of the ψ stencil are equivalent to those after one step of $\mathcal{SF}^k[\psi]$.*

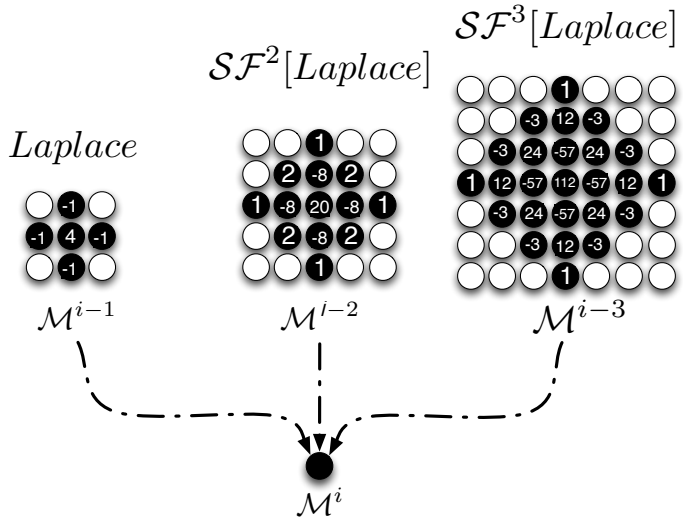


Figure 5.7: Graphical representation of $\mathcal{SF}^i[Laplace]$ stencils for i equal to one to three.

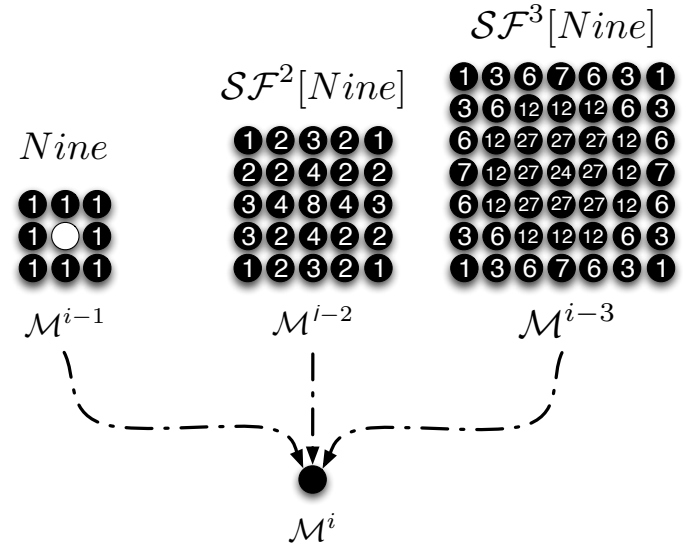


Figure 5.8: Graphical representation of $\mathcal{SF}^i[nine]$ stencils for i equal to one to three.

Proof. The proof comes from the definition of $\mathcal{M}^{i+1}[e]$ in Equation (5.4). □

The previous property asserts that the transformation that we defined behaves in the way we planned: it returns a stencil which merges two or more steps of the original stencil into one.

Theorem 5.2.2 (**$\mathcal{S}$tep $\mathcal{F}$usion Support Transformations are $\mathcal{HUA}$**). *Let ψ and χ be two generic $\mathcal{HUA}$ stencils. $\mathcal{SF}^2_\psi[\chi]$, the result of the support transformation of ψ applied to χ, is a $\mathcal{HUA}$ stencil.*

Proof. We demonstrate that the set $\mathcal{SF}^2_\psi[\mathcal{S}^\chi{}_i]$ can be expressed in the form $e + \{\beta_1, \beta_2, \ldots, \beta_n\}$ where $\forall\alpha\ (e + \beta_\alpha) \in \mathcal{M}$. Exploiting the definition of $\mathcal{S}^\chi_i$ and $\mathcal{S}^\psi_{i-1}$ in the $\mathcal{HUA}$ model we get:

$$\begin{aligned} \mathcal{S}^\chi_i(e) &= \left\{\alpha_1, \ldots, \alpha_n |\ \forall j\ \alpha_j = (e + \beta_{\alpha_j}) \in \mathcal{M}\right\} \\ \forall\alpha \in \mathcal{S}^\chi_i(e),\ \mathcal{S}^\psi_{i-1}(\alpha) &= \alpha + \left\{\delta_1, \ldots, \delta_m\right\} \textit{ where } \forall j\ (\alpha + \delta_j) \in \mathcal{M} \\ &\Downarrow\!= e + \left\{\delta_1 + \beta_\alpha, \ldots, \delta_n + \beta_\alpha\right\} \end{aligned} \tag{5.5}$$

Replacing Equation (5.5) in the definition of $\mathcal{SF}^2_\psi[\mathcal{S}^\chi{}_i]$ (see Equation 5.4), we get:

$$\begin{aligned} &\mathcal{SF}^2_\psi[\mathcal{S}^\chi{}_i](e) = \\ &= \left\{\forall\omega \in \mathcal{M} | \omega \in \bigcup_{\alpha\in\mathcal{S}^\chi_i(e)} \left\{e + \left\{\delta_1 + \beta_\alpha, \delta_2 + \beta_\alpha, \ldots, \delta_n + \beta_\alpha\right\}\right\}\right\} \\ &\Downarrow\!= \left\{\forall\omega \in \mathcal{M} | \omega \in e + \left\{\bigcup_{\alpha\in\mathcal{S}^\chi_i(e)} \left\{\delta_1 + \beta_\alpha, \delta_2 + \beta_\alpha, \ldots, \delta_n + \beta_\alpha\right\}\right\}\right\} \end{aligned}$$

Each $\omega \in \mathcal{SF}[\mathcal{S}_i](e)$ has the form $e + \beta_\omega$; we can conclude that the set $\mathcal{SF}[\mathcal{S}_i](e)$ is compatible with the $\mathcal{HUA}$ model. Because all the remaining elements are compatible with $\mathcal{HUA}$, the demonstration is complete. □

The previous property is essential for the definition of $\mathcal{SF}-transformations$; it asserts that the support transformation of a $\mathcal{HUA}$ stencil maps $\mathcal{HUA}$ stencils onto $\mathcal{HUA}$ stencils. Therefore, first we can conclude the following:

I The series of transformations defined in Equation (5.3) is well defined.

II The result of a $\mathcal{SF}-transformation$ is still a $\mathcal{HUA}$ stencil: results of $\mathcal{Q}-transformations$ can be directly exploited on the transformed stencils.

5.2.3 Step Fusion for Linear Step Functions

A more specific definition of $\mathcal{SF}-transformations$ can be suggested for those stencils whose functions $\mathcal{F}$ are a linear combination of the input parameters.

Exploiting the associative and commutative properties, which are featured by linear computations, the function composition defined by the support transformation can be unrolled and presented as a linear combination of domain element values.

Most of the stencil computations feature linear functions: all stencils used in partial differential equation solvers or in image processing. The same *Jacobi*, which computes the arithmetic mean over a set of four elements, has a function which is a linear combination of the shape element values. Equation (5.2) associated to $\mathcal{SF}^2[Jacobi]$ shows that a stencil that features a linear step function is transformed by $\mathcal{SF}-transformations$ into a stencil that still features a linear step function. The linearity is obviously an invariant with respect to $\mathcal{SF}-transformations$.

Because stencils characterized by linear step function are widespread, we introduce for them a specific definition of the support functions.

Definition 5.2.3 (**$\mathcal{S}$tep $\mathcal{F}$usion Support Transformations for Linear Functions**).

$$
\begin{aligned}
\forall e \in \mathcal{M} \quad & \overset{\mathcal{SF}^2_\psi[\chi_{step_i}]}{\rightarrow} \quad \left(\mathcal{F}^\chi{}_i, \mathcal{F}^\psi{}_{i-1}, \mathcal{SF}^2_\psi[\mathcal{S}^\chi_i]\right) \\
\mathcal{SF}^2_\psi[\mathcal{S}^\chi{}_i] \quad & = \quad \left\{ \forall \omega \in \mathcal{M} | \omega \in \bigcup_{g_\alpha \in \mathcal{S}^\chi_i(e)} \mathcal{S}^\psi_{i-1}(g_\alpha), \right\} \\
\mathcal{M}^{i+1}[e] \quad & = \quad \sum_{\omega_j \in \mathcal{SF}^2_\psi[\mathcal{S}^\chi{}_i]} \lambda_j * \mathcal{M}^{i-1}[\omega_j] \qquad (5.6)
\end{aligned}
$$

It is important to notice that the coefficients λ_j are constants that can be determined statically when calculating the shape of $\mathcal{SF}^k[\psi]$. In the rest of the Chapter we consider only stencils featuring a linear step function.

5.2.4 $\mathcal{SF}$ and Oversending

Summing up what we have seen in the Chapter up to now, we first presented the oversending method stressing that in the literature it is expressed at the concurrent level. Then we presented $\mathcal{SF}-transformations$, transformations defined at the functional level which result in stencils that compute several steps in a single step.

What remains now is a demonstration that the communication pattern implied by a stencil transformed with $\mathcal{SF}-transformations$ coincides with that associated with the oversending method.

Property 5.2.1 (**$\mathcal{SF}-transformations$ and Oversending**). *Let ψ be a $\mathcal{HUA}$ stencil and let $\mathcal{SF}^k[\psi]$ be the result of a $\mathcal{SF}-transformation$ applied to ψ. At the*

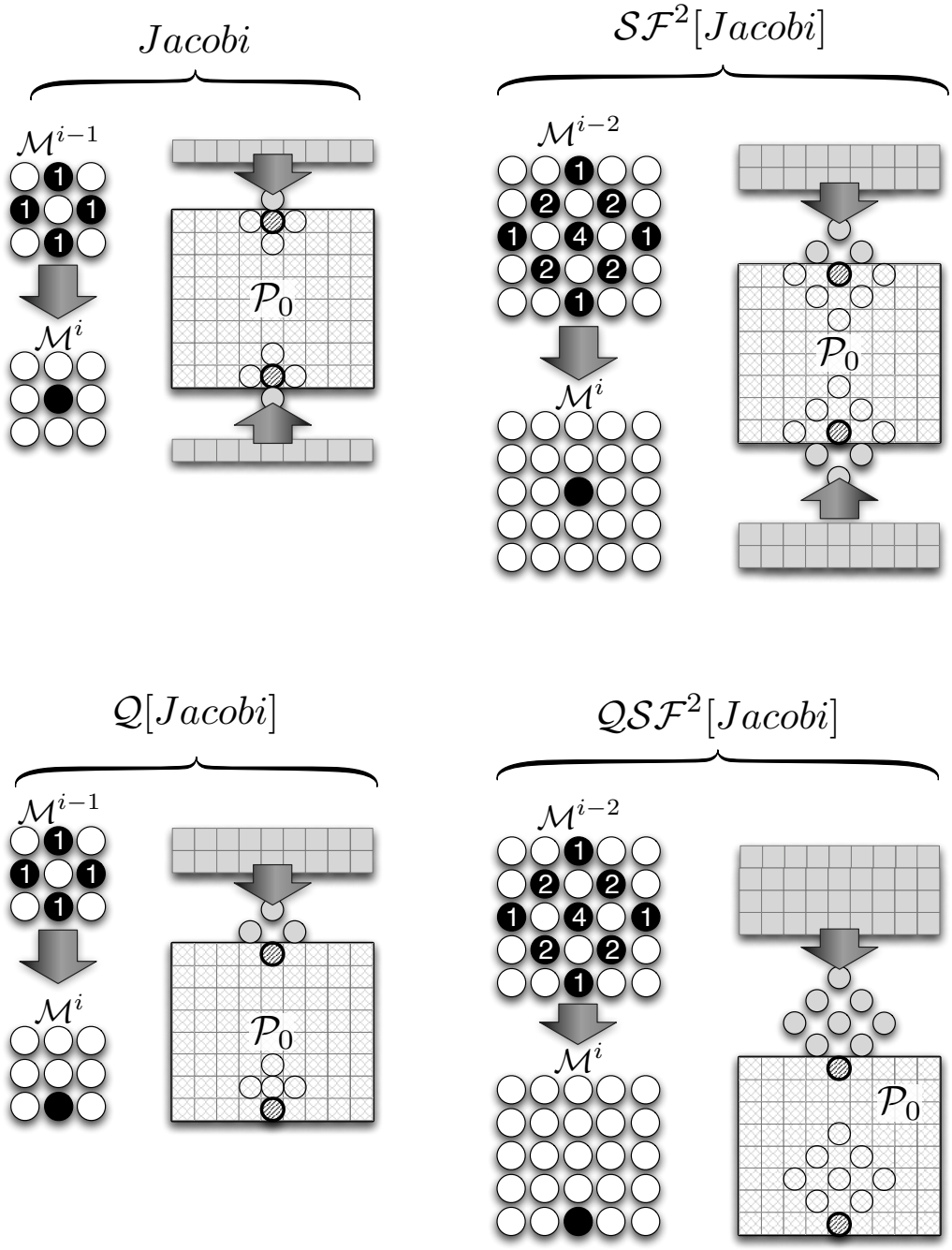


Figure 5.9: Graphical representations of shapes and communication patterns of $Jacobi$, $\mathcal{SF}^2[Jacobi]$, $\mathcal{Q}[Jacobi]$, $\mathcal{QSF}^2[Jacobi]$ stencils.

concurrent level, the implementation of ψ, according to the oversending method of level k, and the implementation of $\mathcal{SF}^k[\psi]$, according to the naive method, feature the same communication pattern.

Proof. Oversending avoids communication for a set of consecutive steps. This means that in the step that features communications, all the information has to be exchanged to update locally the partition for the following $k-1$ steps.

The communications of $\mathcal{SF}^k[\psi]$ imply an information exchange to compute $k-1$ steps of the original stencil.

We can conclude that the information flow is the same in both implementations as is the communication pattern. A graphical proof of this can be derived by comparing the communication patterns reported in Figure 5.9 and Figure 5.5. □

Finally for completeness we define, as we did symmetrically for oversending, a new transformation that mixes both $\mathcal{SF}-transformations$ and $\mathcal{Q}-transformations$. The new transformations, that we call $\mathcal{QSF}-transformations$, feature the same communication performance model as the method that we called $q-oversending$.

5.3 $\mathcal{SF}$ and Sequential Computations

In the previous Section we presented the class of $\mathcal{QSF}-transformations$, where the transformed stencils are found from the original stencils by merging two or more steps into one. In this section we focus the discussion on the benefit of exploiting $\mathcal{QSF}-transformations$ in sequential computations.

The class of $\mathcal{QSF}$–$transformations$, in which a stencil can be transformed with different levels of step fusion, provides different sequential and parallel implementations of the same stencil. For example, one way to implement the *Jacobi* is the classical method, that is exploiting the four element stencil. Another version is the transformed stencil $\mathcal{SF}^2[Jacobi]$, whose single step is equivalent to two steps of the classical *Jacobi*. Other implementations can be defined by raising the level of the step fusion. We are therefore interested in the features of different implementations and in particular we study when a transformed stencil can achieve, in sequential executions, a greater speed up compared to the classical implementation.

We recall that in this Section we consider only stencils featuring a linear step function.

5.3.1 Relation between Shape Cardinality and $\mathcal{SF}$ Level

We start the study of $\mathcal{SF}-transformations$ with an analysis of the relation between the shape cardinality and $\mathcal{SF}$ level. This relation is important because the computation load, as is easy to understand, in most cases depends linearly on the number of elements in the shape.

Table 5.2 reports the number of elements of three stencils analyzed with different levels of step fusion. We remember that by definition we have $\mathcal{SF}^1[\psi] = \psi$.

	Jacobi	*Laplace*	*Nine*
$\mathcal{SF}^1$	4	5	8
$\mathcal{SF}^2$	9	13	25
$\mathcal{SF}^4$	16	25	49
$\mathcal{SF}^4$	25	41	81

Table 5.2: Number of elements of stencils produced by the application of different $\mathcal{SF}-transformations$ to *Jacobi*, *Nine* and *Laplace*

To better understand the trend in the relation between levels and shape elements for different levels of fusion and for different stencils, we refer to Figure 5.10 where

the *Element Increasing Factor* (*EI* factor) is plotted for *Jacobi*, *Nine* and *Laplace*. The *EI* is defined parametrically as a function of the fusion level as:

$$EI_{\psi}(i) = \frac{\left|\mathcal{SF}^{i}[\mathcal{S}^{\psi}]\right|}{\left|\mathcal{SF}^{1}[\mathcal{S}^{\psi}]\right|}$$

In the chart, the dotted line represents the bisector of the first quadrant. We can see that the more the fusion level increases the more the curves move away from the bisector.

Let ψ be one of the three stencils that we have plotted. If the shape of ψ features a cardinality equal to s, then the shape of $\mathcal{SF}^{k}[\psi]$ features a number of elements that is higher than $k * s$.

Recalling that the computation load is linked to the shape cardinality, we can assert that, in order to update the working domain to a certain time, the higher the step fusion level, the more computations are required.

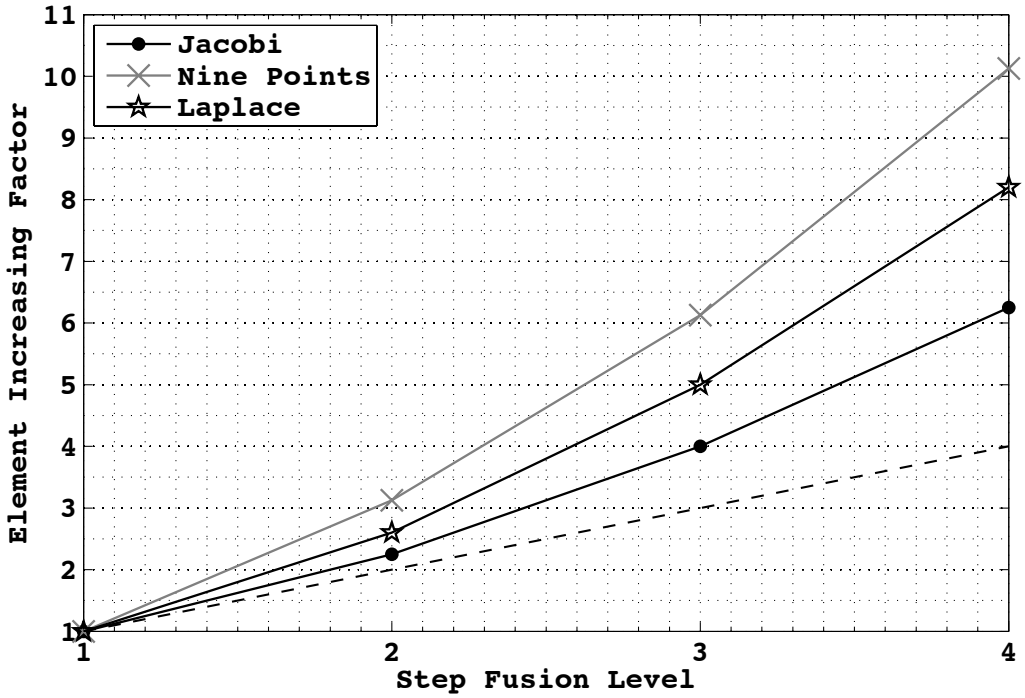


Figure 5.10: Element Increasing Factor for Jacobi, Laplace and Nine.

5.3.2 Temporal Locality Factor in $\mathcal{SF}-transformations$

If the computational overhead increases with increasing level of step fusion, there is another component that has to be taken into account: temporal locality.

A transformed stencil features a different shape, which means that the functional dependencies are changed.

Figure 5.11(a) analyzes the computation of a *Jacobi* along a matrix row. The temporal locality is easy to highlight by considering the effect on registers.

While for the computation of the first and second elements of the row, all the shape elements have to be loaded, for the remaining computations only three elements of the shape can be loaded: indeed one value has already been loaded.

Similarly for $\mathcal{SF}^2[Jacobi]$, after a small transition, the computation saves four load operations. In general, as presented in Figures 5.12(a), 5.12(c) and 5.12(b) for different stencils, the number of reused elements can be established by subtracting from the total number of elements the number of rows on which the stencil is scattered. This number depends linearly on the level of fusion.

We can conclude that there are two important factors that are at stake: both of them increase with the level of step fusion but only one impacts positively on performance. A model to study the results of the two factors is presented in the next Section.

5.3.3 Asymptotic Analysis of Computations and Communications

We focus on a really simple model to predict some properties of the performance trend of transformed stencils. We consider an architectural model with an infinite number of registers, a memory and no cache hierarchy.

Let T^{ψ}_{el}, the mean time required to compute one element according to a stencil ψ, be given by the equation

$$T^{\psi}_{el} = T^{\psi}_{mem} + T^{\psi}_{op}$$

The quantity T^{ψ}_{mem} is the time spent in loading operations while T^{ψ}_{op} is spent on arithmetic operations. T^{ψ}_{mem} and T^{ψ}_{op} are respectively proportional to the number of load operations (N_{load}) and to the number of elements in the stencil shape ($|\mathcal{S}^{\psi}|$). Therefore, we define

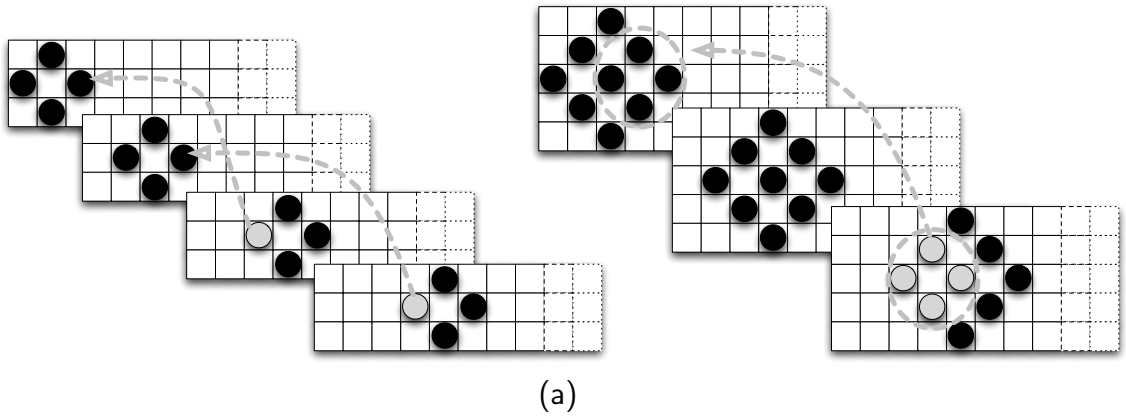


Figure 5.11: Graphical representation of time locality for *Jacobi* and $\mathcal{SF}[Jacobi]$ stencils

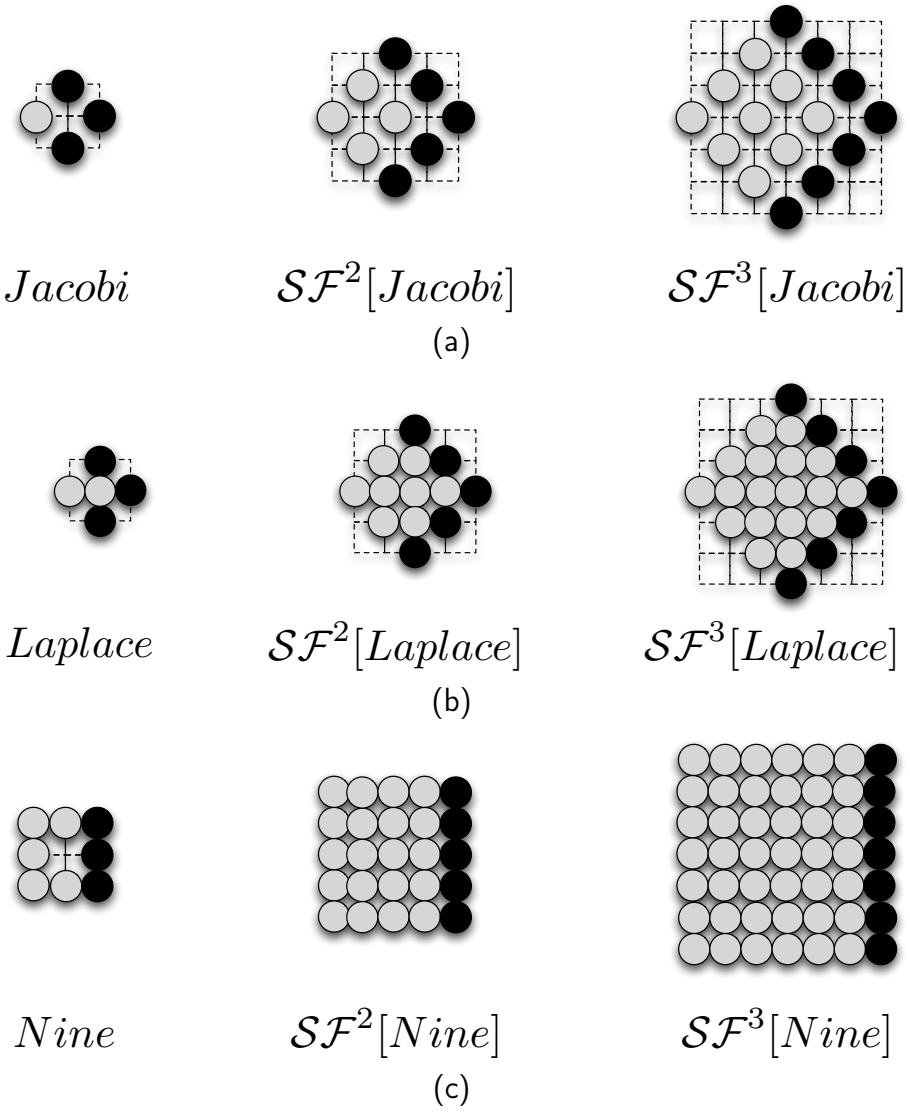
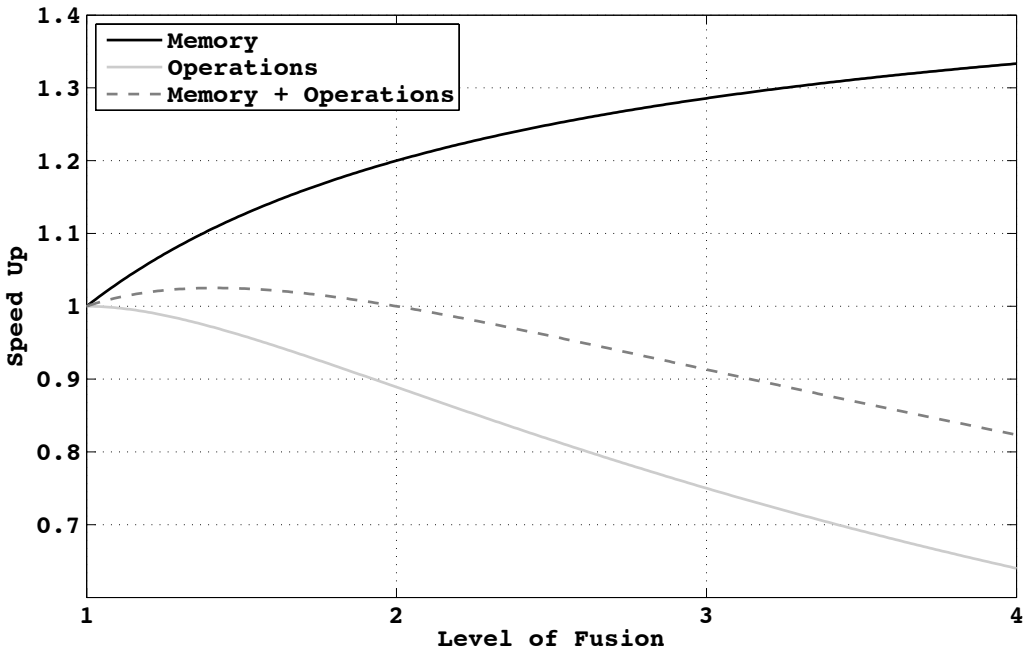
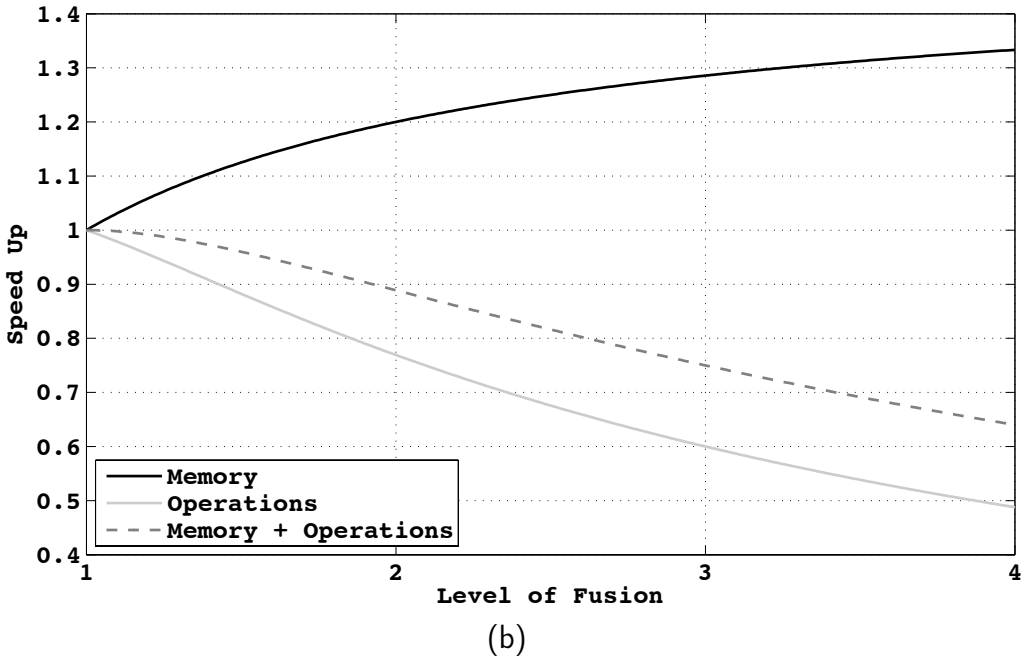


Figure 5.12: Graphical representation of time locality for different stencils



(a)



(b)

Figure 5.13: Performance trend for $\mathcal{SF}-transformations$ of Jacobi 5.13(a) and Laplace 5.13(b)

$$T^{\psi}_{mem} = \lambda * N_{load}$$

$$T^{\psi}_{op} = \kappa * |\mathcal{S}^{\psi}|$$

λ and κ are two constants modelling the impacts, respectively, of load and arithmetic operations on the physical target architecture.

We are not interesting in investigating λ and κ values, our analysis focuses on discovering some trends in performance, leaving the two constant unspecified.

Focusing on *Jacobi* and *Laplace*, we can define, parametrically with respect to the fusion level i, the mean time to compute one element as:

$$T^{Jacobi}_{el}(i) = \lambda * \overbrace{\left[2i+1\right]}^{T^{Jacobi}_{mem}(i)} + \kappa * \overbrace{\left[(i+1)^2\right]}^{T^{Jacobi}_{op}(i)}$$

$$T^{Laplace}_{el}(i) = \lambda * \overbrace{\left[2i+1\right]}^{T^{Lapalce}_{mem}(i)} + \kappa * \overbrace{\left[(i+1)^2 + i^2\right]}^{T^{Laplace}_{op}(i)}$$

We are interested in studying the improvement associated with $\mathcal{SF}$–$transformations$, therefore we analyze the time gain factor of the two computations. We start by analyzing the two components, T_{op} and T_{mem}, separately. The advantage of this strategy is that the λ and κ variables disappear in the time gain formula.

We recall that we have defined the time gain as the time of a reference computation divided by the time of an equivalent computation that we wish to study. A time gain higher than one implies a gain in performance while the opposite implies a performance loss.

In this analysis, we consider as a reference the time required to compute one step of the original stencil (T^{ψ}). We wish to compare the computation of ψ with those of the stencils $\mathcal{SF}^i[\psi]$ ($T^{\mathcal{SF}^i[\psi]}(i)$). Because in one step $\mathcal{SF}^i[\psi]$ computes i steps of the original stencil we normalize the time dividing it by i, therefore the time gain formula is given by:

$$\mathcal{G}^{\psi}_{mem}(i) = \frac{i * T^{\psi}(1)}{T^{\mathcal{SF}^i[\psi]}(i)}$$

Coming back to separate analyses of the memory and computation impacts, we can assert the following points.

I The time gain associated with only load operations is defined as:

$$\mathcal{G}^{Jacobi}_{mem}(i) = \frac{i * T^{Jacobi}_{mem}(1)}{T^{Jacobi}_{mem}(i)} = \frac{3 * i}{2i+1}$$

$$\mathcal{G}^{Laplace}(i) = \frac{i * T^{Laplace}_{mem}(1)}{T^{Laplace}_{mem}(i)} = \frac{3 * i}{2i + 1}$$

because, as was highlighted before, the number of loads is equivalent to the number of rows on which the stencil shapes are spread. Both *Jacobi* and *Laplace* feature the same time gain for load operations.

With an increase in the level of step fusion, the time gain increases (as plotted in Figures 5.13(a) for the Jacobi case and in Figure 5.13(b) for the Laplace). The curves are labelled with the name *memory* increase as well but present an horizontal asymptote: it never exceeds the value 3/2 as demonstrated by the limit:

$$\lim_{i\to\infty} \mathcal{G}^{Jacobi}_{mem}(i) = \lim_{i\to\infty} \mathcal{G}^{Laplace}_{mem}(i) = \frac{3}{2}$$

II On the other hand, the time gain related to arithmetic operations is defined as:

$$\mathcal{G}^{Jacobi}_{op}(i) = \frac{i * T^{Jacobi}_{op}(1)}{T^{Jacobi}_{op}(i)} = \frac{4 * i}{(i + 1)^2}$$

$$\mathcal{G}^{Laplace}_{op}(i) = \frac{i * T^{Laplace}_{op}(1)}{T^{Laplace}_{op}(i)} = \frac{5 * i}{(i + 1)^2 + i^2}$$

In contrast with the other case, the two components are different for the *Jacobi* and *Laplace* stencils. Their curves, labelled as *operations* are reported in Figures 5.13(a) and Figure 5.13(b).

As analyzed previously, with an increase in the level of step fusion, the computational load is heavier and therefore the time gain is always less than one. More precisely, it approaches zero monotonically as $O(1/i)$ as demonstrated by the following limits.

$$\lim_{i\to\infty} \mathcal{G}^{Jacobi}_{op}(i) = \lim_{i\to\infty} \frac{4}{i}$$

$$\lim_{i\to\infty} \mathcal{G}^{Lapalce}_{op}(i) = \lim_{i\to\infty} \frac{5}{2i}$$

Without investigating specific values of λ and κ for a target architecture, we now focus on the asymptotic trend of $\mathcal{G}_\psi$:

$$\lim_{i\to\infty} \mathcal{G}^{Jacobi}_{\psi} = \lim_{i\to\infty} \frac{T^{Jacobi}_{\psi}(1) * i}{T^{Jacobi}_{\psi}(i)} = \lim_{i\to\infty} \frac{3\lambda + 4\kappa}{\kappa i} = o\left(\frac{1}{i}\right)$$

$$\lim_{i\to\infty} \mathcal{G}_{\psi}^{Laplace} = \lim_{i\to\infty} \frac{T_{\psi}^{Laplace}(1) * i}{T_{\psi}^{Laplace}(i)} = \lim_{i\to\infty} \frac{3\lambda + 5\kappa}{(\kappa+1)i} = o\left(\frac{1}{i}\right)$$

From the definition of limit, we can conclude, independently of the value of λ or κ, that there is a level $\widetilde{i}$ of step fusion beyond which there is no improvement in performance. Both λ and κ impact on the value of $\widetilde{i}$.

We therefore have two possible scenarios.

I $\widetilde{i}$ can be equal to one, which means that there is no performance improvement by exploiting $\mathcal{SF}$ for sequential kernels.

II $\widetilde{i}$ can be greater than one, that is $\mathcal{SF}$ can be used to lower the computation time.

To visualize the trend of $\mathcal{G}_{\psi}$, we have reported in Figure 5.13(a) and Figure 5.13(b) the curves corresponding to the case $\kappa = \lambda = 1$.

It is important to notice that in our analysis we did not consider any limits on the number of registers in the architecture.

5.3.4 Taking into account Cache Memory Hierarchy

The temporal locality we analyzed in the previous Section has an important impact on the caches memory hierarchy. The principle is the same that we analyzed for register temporal reuse, but the explanation is slightly more complex, which is why we started the analysis of temporal locality from the register perspective.

In this Section we do not aim to give an in-depth analytic analysis of the temporal locality impact on caches, as we did previously in the case of register reuse. We simply wish to give a proof that the temporal locality introduced by a higher level of step fusion can influence also the performance of caches.

We consider a simple model with only one level of caching between the memory and registers. We suppose each cache line to be of length κ and for the sake of explanation we do not consider any bounds on the number of lines. Indeed, we consider that the number of lines is sufficient to exploit completely the benefit of temporal locality.

We analyze first the effects of the *Jacobi* stencil on caching and then we compare them with those of $\mathcal{FS}^2[Jacobi]$. More precisely, we compare one step of $\mathcal{FS}^2[Jacobi]$ with two steps of *Jacobi*.

In our analysis we take as a reference the sequential naive implementation; the method that features two matrices, one for the input data and one for the output data. We suppose that the matrices have a number of rows equal to the variable λ and that each row is equivalent in size to a number of η cache lines.

On average, an element waiting to be computed according to the Jacobi stencil requires that three cache lines be previously filled up with the correct portions of

the input matrix. Hence, to compute the entire matrix once, we can say that the number of cache lines that are read is approximately equal to:

$$3 * \lambda * \eta$$

A second iteration of the Jacobi would require the same number of cache line read operations. In conclusion, in order to complete the computation of two steps, the total number of cache lines that are read is approximately equal to:

$$6 * \lambda * \eta$$

If we consider one step of the stencil $\mathcal{SF}^2[Jacobi]$, which as we recall is computationally equivalent to two computation steps of $Jacobi$, the total number of cache lines that are read is approximately equal to:

$$5 * \lambda * \eta$$

Therefore, in order to update the working domain according to the computation of two iterations of the Jacobi stencil, the implementation which exploits the $\mathcal{SF}-transformations$ performs approximately $\lambda * \eta$ cache line read operations less than the naive sequential implementation.

5.3.5 Experimental Results

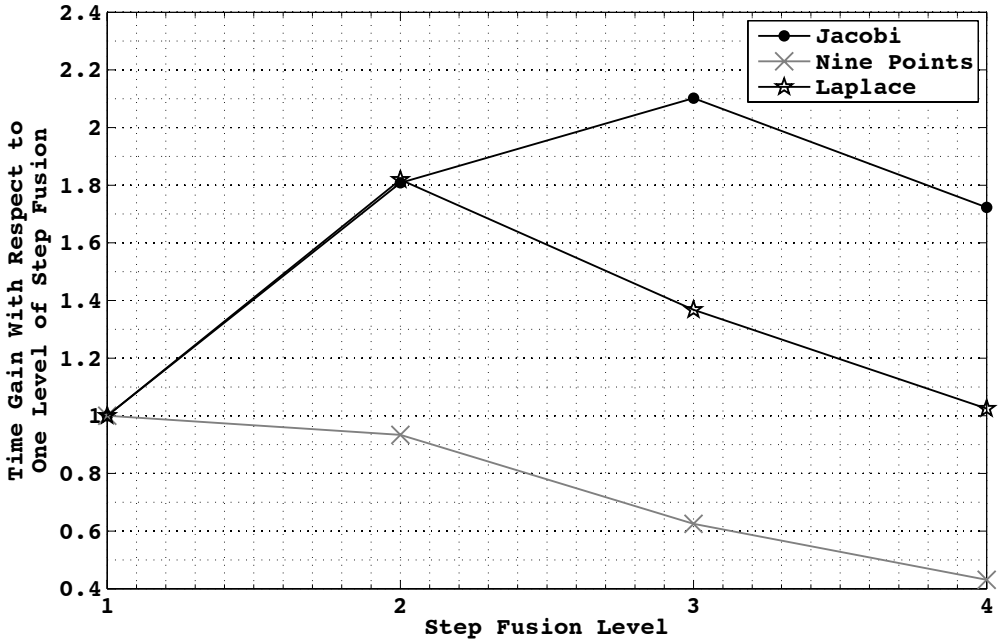
We have evaluated experimentally the time gain of different levels of step fusion, considering a wide range of cases. We examine the three example stencils $Jacobi$, $Laplace$ and $Nine$ and four target architectures featuring different processor frequencies and cache sizes.

The tests were run targeting a size of the working domain that could not completely fit the cache memory levels. For the two-dimensional case, we target matrices of 72 MB. The results achieved are presented in the charts of Figures 5.14(a), 5.15(a), 5.16(a), 5.17(a).

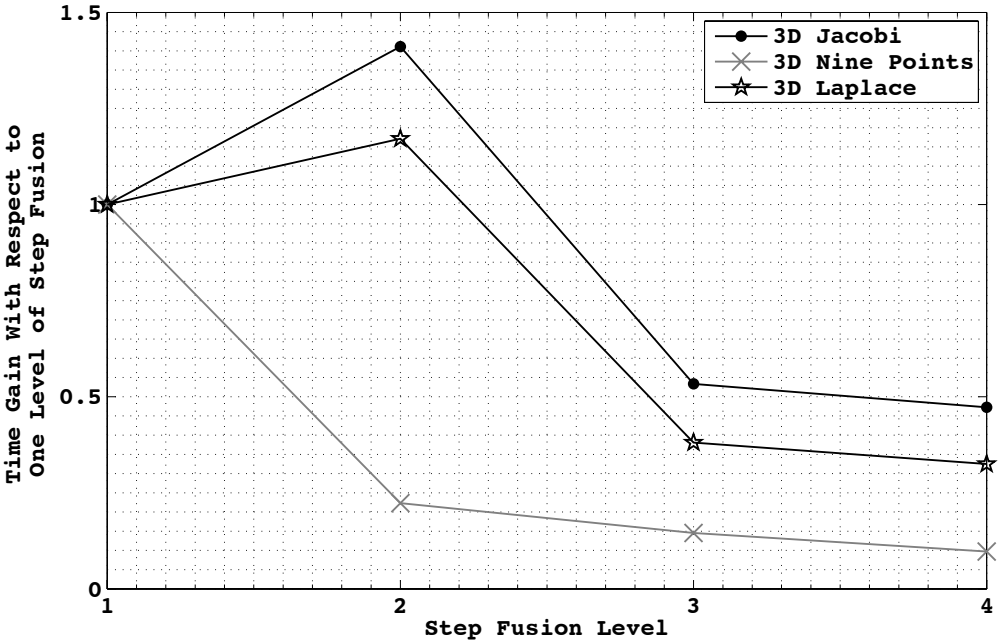
For the $Nine$ stencil, which features a larger increase in the number of operations associated with the rise in step fusion level, no benefits were detected by the use of $\mathcal{SF}-transformations$. This means that the impact of the arithmetic overhead is greater than reuse effects.

The results for the other two stencils are different. In all the targeted architectures, $SF^2[Jacobi]$ and $SF^2[Laplace]$ offer better performance than the original stencils. In some cases, the time gain is close to two: 2.1 for $Jacobi$ and 1.8 for $Laplace$. Therefore the sequential implementation of the transformed stencil is nearly two times faster than the original sequential version.

Some tests were also performed using the three-dimensional extension of Jacobi, Laplace and Nine. The charts in Figures 5.14(b), 5.15(b), 5.17(b) show the results. For the three-dimensional cases we selected a working domain of 216MB.

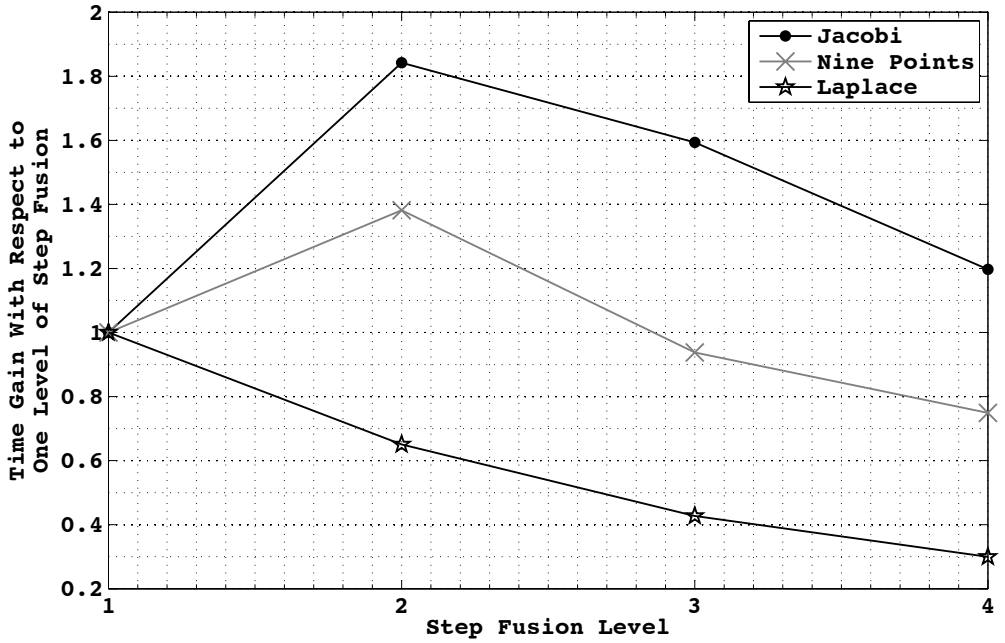


(a)

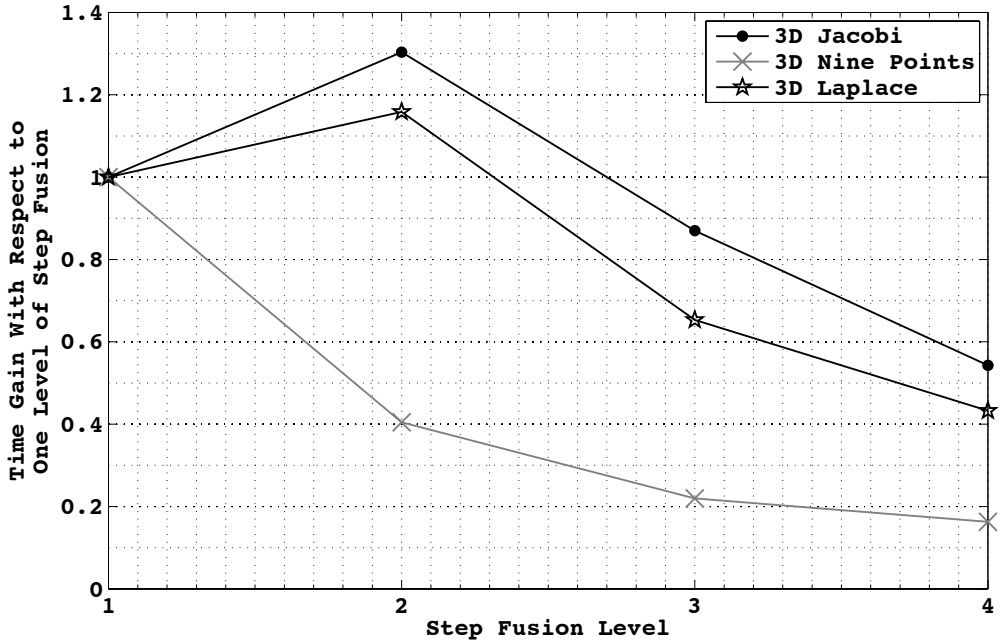


(b)

Figure 5.14: Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB

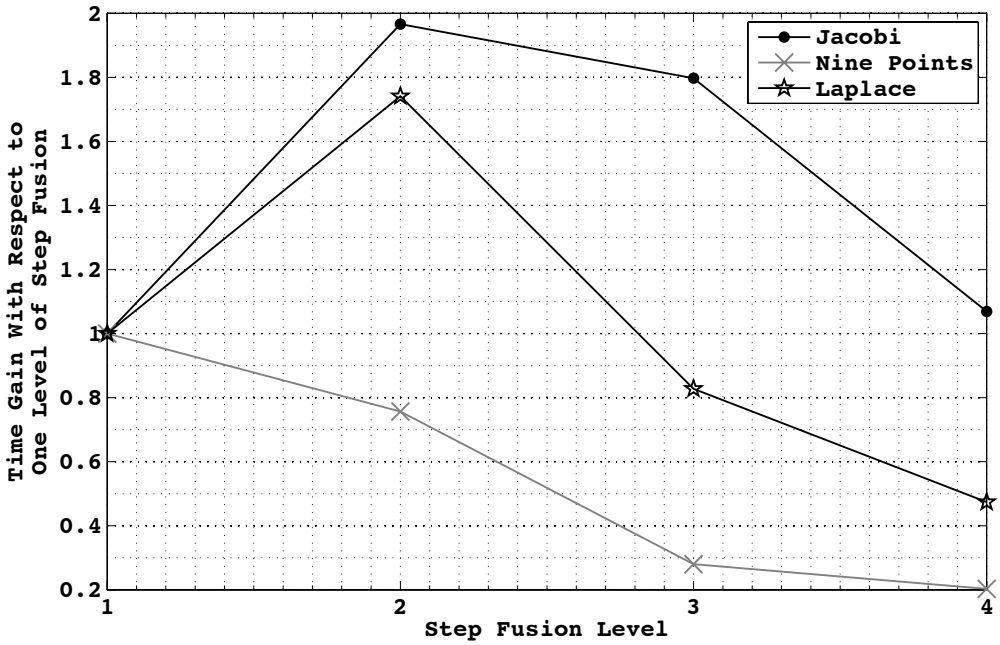


(a)



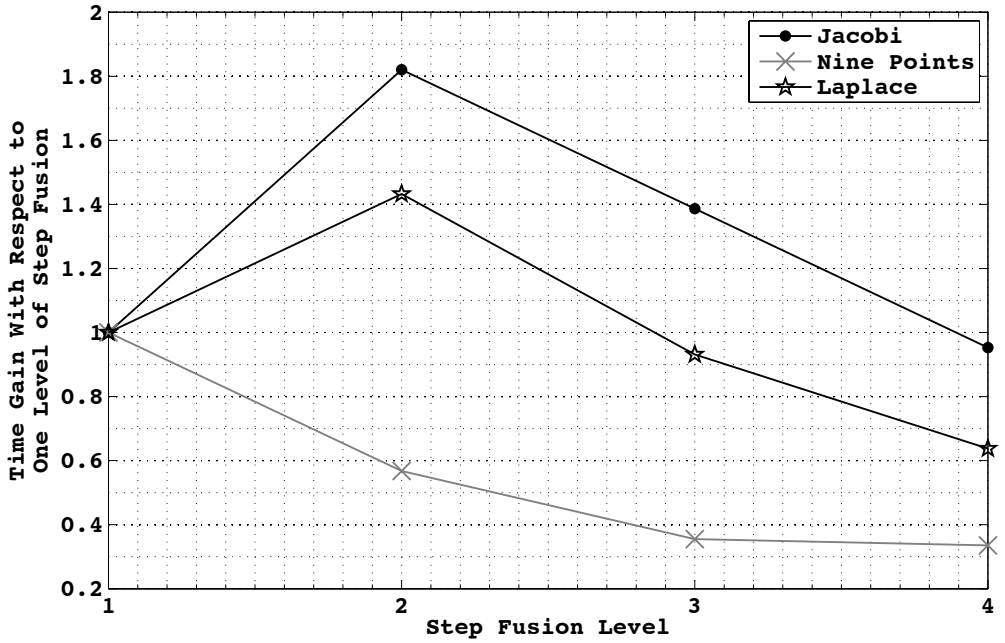
(b)

Figure 5.15: Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB

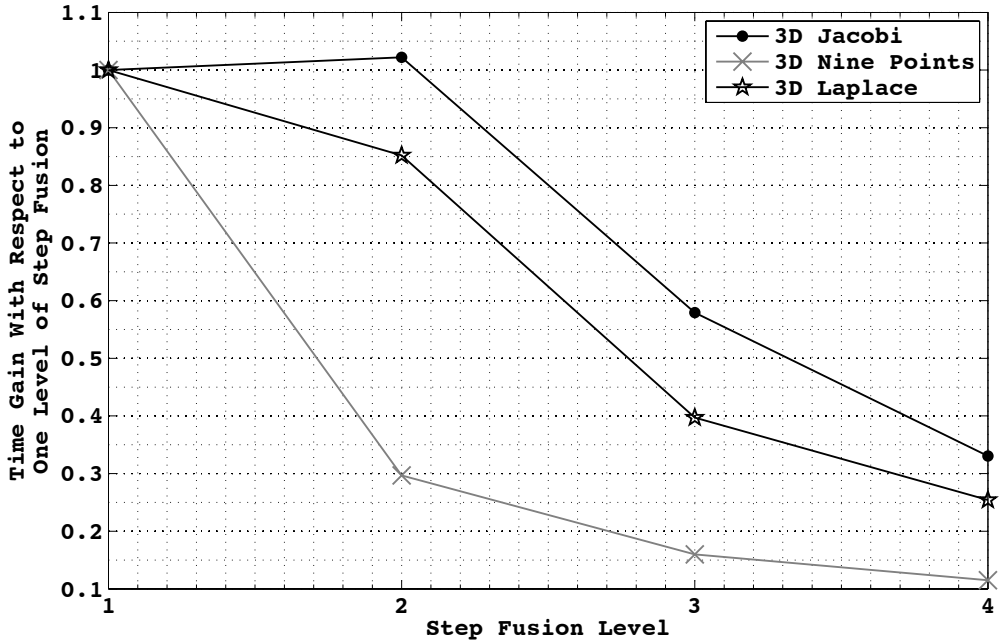


(a)

Figure 5.16: Intel(R) Xeon(R) CPU 5150 @ 2.66GHz cache size 4096 KB



(a)



(b)

Figure 5.17: Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB

The results are slightly different from the ones for the two-dimensional cases. The registered time gains are reduced compared with the previous gains. In the test configuration, the three-dimensional Laplace offers, for a level of step fusion greater than one, a time gain that is less than one; which means a loss of performance. This behaviour is caused by an increasing impact of the computational component compared to the other.

In support of the previous assertion, we report in Table 5.3 the number of shape elements featured by the stencil in a three-dimensional space and in Figure 5.18 a chart of the Element Increasing Factor.

	Jacobi	*Laplace*	*Nine*
$\mathcal{SF}^1$	6	7	26
$\mathcal{SF}^2$	19	25	125
$\mathcal{SF}^4$	44	63	343
$\mathcal{SF}^4$	85	129	729

Table 5.3: Number of elements in the stencils resulting from the application of different $\mathcal{SF}-transformations$ to the three-dimensional extensions of *Jacobi*, *Nine* and *Laplace*

5.3.6 Conclusions

The previous analysis, made for two-dimensional cases, are confirmed by the experimental results: all the measured speed increases for a step fusion transformation above the second level are monotonically decreasing. We conclude that sequential code optimized with $\mathcal{SF}-transformations$ and a low level of fusion can provide in some cases an outstanding performance improvement. The improvement is reduced when targeting three-dimensional spaces.

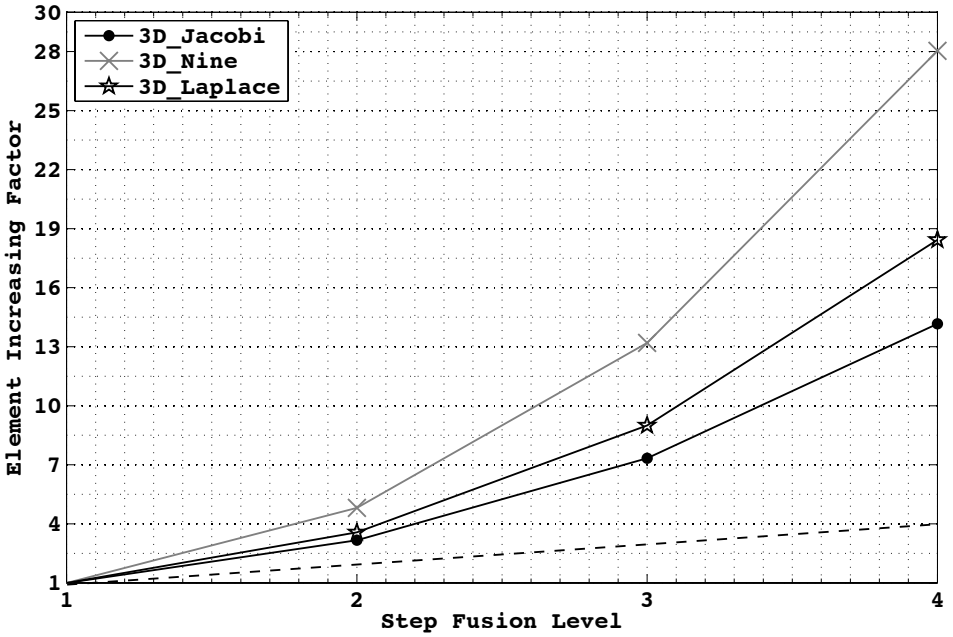


Figure 5.18: Element Increasing Factor for Jacobi, Laplace and Nine extended to a three-dimensional space.

Chapter 6

Space Overlapping Transformations

Abstract

In the previous Chapters, we focused on techniques for optimizing both communication and computational overheads, indeed for the two problems we introduced $\mathcal{Q}-transformations$ and $\mathcal{SF}-transformations$, respectively. In this Chapter, we concentrate on the memory requirements of the implementations at the concurrent level of a $\mathcal{HUA}$ stencil.

Implementations of a $\mathcal{HUA}$ stencil which approximately halve the memory requirement to represent the working domain are well known, but they suffer from the drawback of increasing the computational overhead. Indeed, these techniques require a copy operation for each element of the working domain.

In this Chapter, we present and formally prove the existence of specific $\mathcal{Q}-transformations$ called $\mathcal{QM}-transformations$. The stencils resulting from these new transformations can be associated to an *in-situ* implementation that halves the memory requirements without introducing other computational overheads.

Contents

6.1 Implementation of the Working Domain

We have seen in previous Chapters the importance of the working domain component in both the structured and $\mathcal{HUA}$ models.

At the functional level, the working domain models statically the complete evolution of the stencil computation, thanks to the temporal dimension associated with the evaluation map. Indeed, the evaluation of an element is possible in any computation time because it results in a static parametric formula.

At the concurrent level, the aim of the data structures that implement the working domain is to store, at the end of the last step of the application, the final values of all spatial structure elements.

During the run time computation, the temporal dimension, which is featured by the working domain at the functional level, is lost and only the values at the current instant are stored. We would like to highlight that the implementation of the working domain includes both the definition of data structures for the values and strategies for updating the values from the beginning to the end of the one step computation.

For the sake of simplicity, all the points that we make in the rest of the Chapter focus on the implementation of a sequential program. Nevertheless, all the results can be reused to implement a partition of the working domain.

In the following, we first present and analyze two standard implementations of the working domain. Then we pass on to an introduction of a new implementation strategy which, as will be clear after a deeper analysis, exploits $\mathcal{Q}$–$transformations$. The changes that the transformations make to the stencil shape are such that it is possible to minimize the memory requirements without introducing any copy overheads.

To present the three different implementations, we refer once again to the monodimensional Jacobi stencil.

6.1.1 Naive Implementation

The easiest way to implement a working domain is to exploit two different data structures. One stores the working domain values at time i and the other will store the values at time $i+1$. We report in Figure 6.1 the pseudo-code of the algorithm. This is the same strategy we used in all previous examples. This method does not introduce any constraint on the visit pattern of the data structures.

In order to analyze the presented strategy from the point of view of memory requirements, let us consider $|\mathcal{M}_{Jacobi}|$ the number of elements in the spatial structure and $d(\mathcal{D}_{Jacoibi})$ the function that returns the size of a value of the computational domain. Finally, modelling the memory requirements with the parameter MR we can claim the following formula.

$$MR_{naive} = 2 * |\mathcal{M}_{Jacobi}| * d(\mathcal{D}_{Jacoibi})$$

```
double J_in[100], J_out[100];                                   1
load_working_domain_values(J_in);                               2
for(i_step = 0; i_step < 4; i_step++){                          3
                                                                4
        forall(x, ∈ J_in){                                      5
                J_out[x] = (J_in[x + 1] + J_in[x − 1])/2;       6
         }                                                      7
         swap(J_in, J_out);(                                    8
}                                                               9
return_working_domain(J_out);                                   10
```

Figure 6.1: Description in pseudo-code of a naive implementation of the Jacobi stencil on a toroidal space.

The memory allocated for the implementation of the working domain is equal to two times that required to store all values of the spatial structure elements.

6.1.2 Support Buffer Implementation

During a step computation i, when an element value at time i is not needed any more for the computation of some value at time $i+1$, its memory can be reused to stores values at time $i+1$. In other words, we are free to target *in-situ* computation strategies, which reuse the input data structures in a step to store output values.

We can consider an *in-situ* computation if we address the following points.

I A specific visit for the updating of the working domain elements is exploited.

II Some support buffers are introduced in order to resolve the data dependencies which are the result of the introduction of *in-situ* computations.

We report in Figure 6.2 the pseudo-code of the algorithm that exploits a standard strategy for *in-situ* computations.

The underlying concept is to compute all updates *in-situ* and to rely on support buffers to save the necessary values that would be lost. We call the presented method for implementing the working domain the buffered strategy.

We wish to highlight three important points of the buffered strategy.

I The dimension of the support buffers depends directly on the geometry of the stencil shape. The more the shape area is extended, the larger is the buffer space.

II All the elements of the matrix are copied. The strategy therefore implies a complete copy of all the elements of the working domain.

```
double J[100];                                                          1
double buffer_support_head;                                             2
double buffer_support_present;                                          3
double buffer_support_future;                                           4
                                                                        5
load_working_domain_values(J);                                          6
                                                                        7
for(i_step = 0; i_step < 4; i_step++){                                  8
        copy(buffer_support_head, J[0];)                                9
        copy(buffer_support_present, J[-1];)                            10
                                                                        11
      for(x = 0; x < 999; x++){                                         12
            copy(buffer_support_future, J[x];                           13
            J[x] = (J[x + 1] + buffer_support_present)/2;               14
            swap(buffer_support_present,                                15
                  buffer_support_future);                               16
        }                                                               17
                                                                        18
      J[-1] = (buffer_support_head+                                     19
                  buffer_support_present)/2;                            20
}                                                                       21
return_working_domain(J);                                               22
```

Figure 6.2: Description in pseudo-code of a buffered implementation of the Jacobi stencil on a toroidal space.

III The copy associated with $buffer\_support\_head$ is mandatory to break the toroidal feature. In the case of stencils defined on non-toroidal spaces this copy is not necessary.

Analyzing the memory requirements of the algorithm, we can extract the following formula.

$$MR_{buffered} = \left(|\mathcal{M}_{Jacobi}| + \phi_{buffer}(S^{jacobi})\right) * d(\mathcal{D}_{Jacoibi})$$

The function ϕ_{buffer} returns for a target stencil shape the size of the required buffers.

Because one of our working hypotheses is that the size of a working domain is considerably wider than the size of the shape, we can consider the buffer size to be negligible in terms of memory requirement.

We conclude that the buffered method, at the price of introducing a computational overhead per step associated with the copying of all the values, halves the memory requirements compared to the naive implementation. We can therefore claim the following formula

$$MR_{buffered} \cong |\mathcal{M}_{Jacobi}|$$

6.1.3 Space Overlapping Implementation

In this Section, we present a specification of $\mathcal{Q}-transformations$ that, at the functional dependency level, transforms the stencil into a relaxed-equivalent one. The resulting stencil can be implemented with the same memory requirements as the buffered implementation but avoiding the copy overhead for the whole working domain.

For the sake of clarity, we present and analyze first the algorithm for the Jacobi case and then highlight from its structure the most important aspects that can lead us to the definition of what we will call $\mathcal{QM}-transformations$.

Figure 6.3 reports the pseudo-code of the new strategy for implementing the working domain. The program is more simple than it could appear at first sight. Indeed, if we replace V^J_{odd} and V^J_{even} with J_{in} and J_{out}, respectively, apart from the visit pattern, the program is similar to the naive one. In each step, values are read only from one vector and written to the other.

By the way, V^J_{odd} and V^J_{even} are not real vectors. We define them as **virtual vectors**, i.e. vectors that are mapped onto other vectors. Virtual vectors can be seen to be index renaming mechanisms.

Both V^J_{odd} and V^J_{even} are mapped onto the same vector J. The first virtual vector is mapped onto the index interval $[0, 99]$, while the second is mapped onto $[1, 100]$. A graphical representation of the overlapping is reported in Figure 6.4(a).

Because in the mapping, the two virtual vectors are partially overlapped, we are in a configuration where loop-carried dependencies are present inside a single step. It is extremely important that the visit of the two virtual vectors guarantees the program semantics.

```
double J[100 + 1];                                                          1
VIRTUAL V^J_even[100]->J[0,99]                                              2
VIRTUAL V^J_odd[100]->J[1,100]                                              3
double buffer_support;                                                      4
                                                                            5
load_working_domain_values(J);                                              6
                                                                            7
for(i_step = 0; i_step < 4; i_step++){                                      8
  if(i%2 == 0){                                                             9
    copy(buffer_support, V^J_odd[-1];)                                      10
    for(x = 99; x > 1; x--){                                                11
      V^J_even[x] = (V^J_odd[x + 1] + V^J_odd[x - 1])/2;                    12
    }                                                                       13
    V^J_even[x] = (J[x + 1] + buffer_support)/2;                            14
  }                                                                         15
                                                                            16
  if(i%2! = 0){                                                             17
    copy(buffer_support, V^J_odd[0];)                                       18
    for(x = 0; x < 99; x++){                                                19
      V^J_odd[x] = (V^J_even[x + 1] + V^J_even[x - 1])/2;                   20
    }                                                                       21
     V^J_odd[x] = (J[x + 1] + buffer_support)/2;                            22
  }                                                                         23
                                                                            24
}                                                                           25
return_working_domain(V^J_odd);                                             26
```

Figure 6.3: Description in pseudo-code of an implementation with space overlapping of the Jacobi stencil defined over a toroidal space.

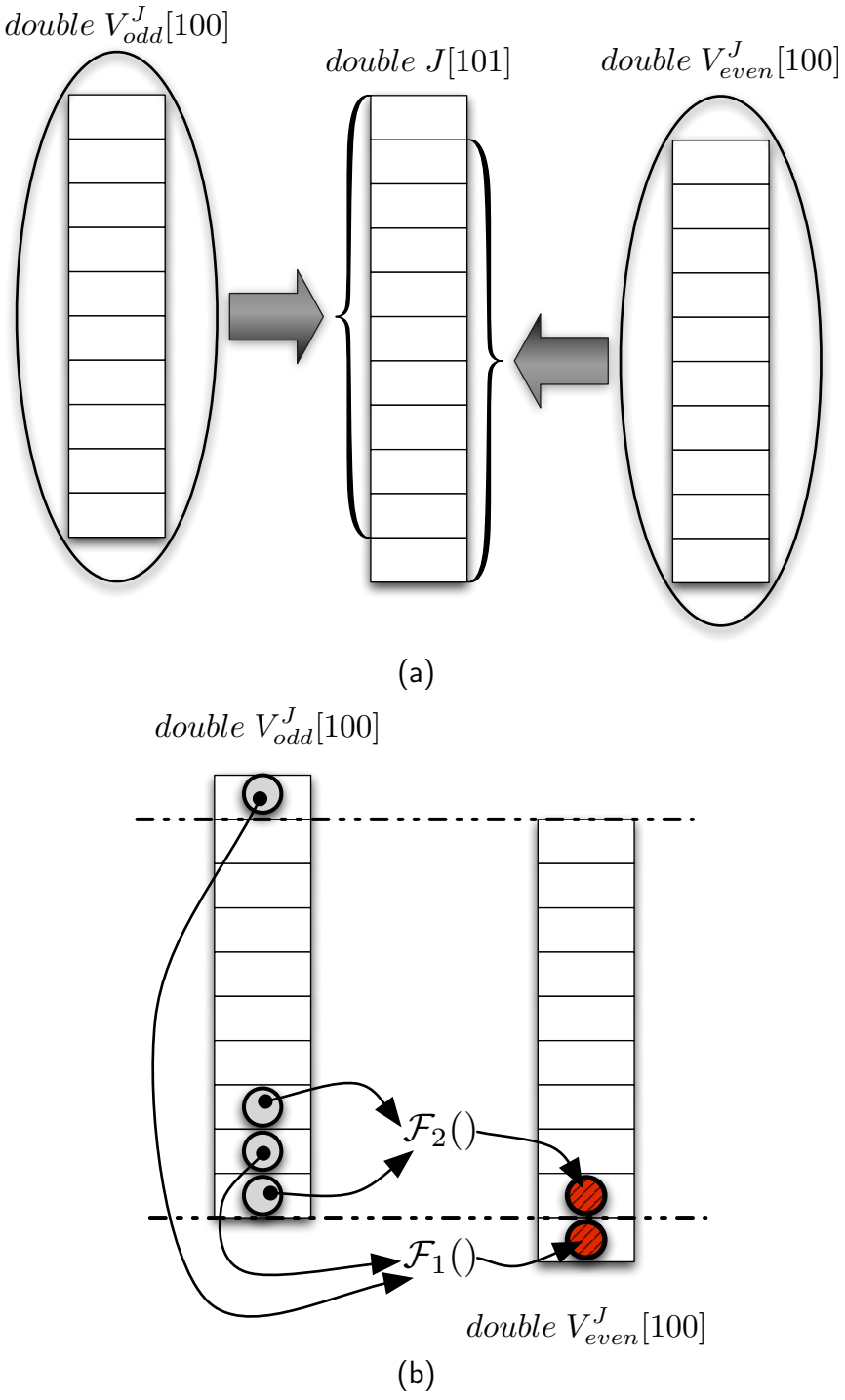


Figure 6.4: Mapping between virtual and real vectors of the overlapping implementation

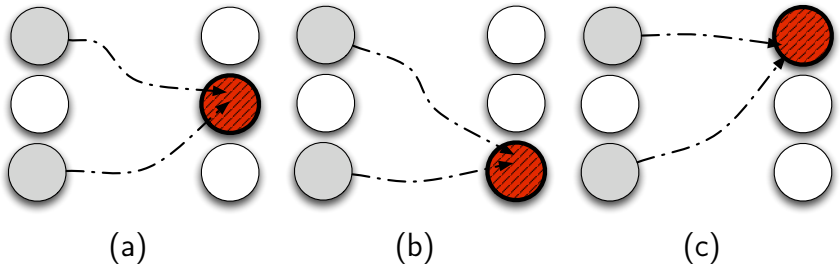


Figure 6.5: $Jacobi$, $\mathcal{Q}^{+}[Jacobi]$, $\mathcal{Q}^{-}[Jacobi]$ shapes.

The two visits which are implemented in the algorithm, one for odd steps and one for even steps, avoid any conflict between read and write operations. Indeed, it is easy to see that all the element storing values at time i are used before they are rewritten with the new values at time $i+1$.

Figure 6.4(b) gives a graphical representation of the previous assertion for odd steps. Following the sequence of operations forced by the visit pattern, we can register what follows.

I The value of the last element of the vector V^J_{odd}, which stores element values at time i, is used to compute the value of the next to last element of V^J_{even}.

II The calculated value is stored in the next to last position of V^J_{even}. Because of the overlap of the virtual vectors, the store operation also changes the value of the last element of V^J_{odd}.

III No other access for read operations is performed on the last element V^J_{odd}.

What we claimed for read and write operations on the last element of the V^J_{odd} vector during odd steps can be extended to all the elements of the vector and also symmetrically to all the elements of V^J_{even} during even steps.

Because we asserted that an element which stores a value at time i is rewritten with a value at time $i+1$ only when the previous value is no longer necessary for the stencil computation, we have proved that the previous algorithm respects the semantics of the computation.

From the perspective of memory requirements, we can model the results obtained for the Jacobi example with the following formula.

$$MR_{\mathcal{QM}} = \left(|\mathcal{M}^{ext}_{Jacobi}| + \phi_{overlapping}(S^{jacobi})\right) * d(\mathcal{D}_{Jacoibi})$$

The function $\phi_{overlapping}$ returns for a target stencil shape the size of the buffer required to break the toroidal constraint. As for the previous implementation, the buffer size is negligible compared to the total size of the working domain. The component $\mathcal{M}^{ext}_{Jacobi}$ represents the extended working domain.

In terms of memory requirements, the method just presented and the previous method are equivalent. The extended dimension of the working domain compensates

for the reduction in buffer size. We can finally claim that this last implementation, without introducing copies of the values of the entire spatial structure, halves the memory requirements. Formally, we can claim the following formula.

$$MR_{\mathcal{QM}} \cong |\mathcal{M}^{ext}_{Jacobi}| * d(\mathcal{D}_{Jacoibi})$$

Let us focus once again on the two virtual vectors and on their relation to stencil shape and point of application. According to the relative indices of each of the two virtual vectors, the shape and point of application are those defined by the original Jacobi stencil as reported in Figure 6.5(a). Nevertheless, if we take into consideration the absolute reference system of the non virtual vector J, the shapes of the odd and even steps are the result of $\mathcal{Q}-transformations$ as shown in Figures 6.5(b) and 6.5(c). The implementation of the working domain that halves the memory requirements can be modelled as a $\mathcal{Q}-transformation$.

6.2 $\mathcal{QM}-transformations$

In this Section we give a formal definition of $\mathcal{QM}-transformations$: specific $\mathcal{Q}-transformations$ that allow the implementation of a program at the concurrent level that is optimized in order to reduce memory requirements. Indeed, we prove that the transformed stencil along with a specific visit pattern can be implemented in a program at the concurrent level that halves the memory requirements without introducing a copy overhead.

As in the case of $\mathcal{Q}-transformations$ for the reduction of communications, $\mathcal{QM}-transformations$ are based on two symmetric transformations: one that will be associated with odd steps and one associated with even steps. We present and prove the positive $\mathcal{QM}-transformations$ and we demonstrate that with the correct pattern of access it is possible to implement an odd step featuring *in-situ* computations.

6.2.1 Positive $\mathcal{QM}-transformation$

Definition 6.2.1 (**Positive** $\mathcal{QM}-transformation$)**.** Let ψ be a $\mathcal{HUA}$ stencil. A positive $\mathcal{QM}-transformation$ is a $\mathcal{Q}-transformation$ that transforms ψ into the $\mathcal{HUA}$ stencil $\mathcal{QM}^{+}[\psi]$ which is equivalent to the original one, except for the step model defined as follows:

$$
\begin{array}{rcl}
\forall e \in \mathcal{M} & \overset{\mathcal{QM}^{+}[\psi]}{\rightarrow} & \left(\mathcal{F}^{\psi}, \mathcal{S}^{\mathcal{Q}^{+}[\psi]}\right) \\
q_m^{+} & = & < q_1^{+}, 0 \ldots, 0 > \\
q_{limit}^{+} & = & -min\left\{\beta_\alpha * \epsilon_1 \forall \beta_\alpha \in \mathcal{R}^{\psi}\right\} \\
q_1^{+} & = & \begin{cases} q_{limit} \textbf{if } (\beta_\alpha * \epsilon_1 = -q_{limit}^{+}) \Rightarrow (\forall j > 1 \ \beta_\alpha * \epsilon_j = 0) \\ \\ q_{limit}^{+} + 1 \text{ in the other cases} \end{cases} \\
\mathcal{R}^{\mathcal{QM}^{+}[\psi]} & = & \mathcal{R}^{\psi} + q_m^{+} \\
\mathcal{S}^{\mathcal{QM}^{+}[\psi]} & = & e + \mathcal{R}^{\mathcal{Q}^{+}} \\
 & \overset{\Downarrow}{=} & e + \left\{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_\alpha = \beta_\alpha + q^{+}\right\} \\
\mathcal{M}^{i+1}_{\mathcal{QM}^{+}[\psi]}[e] & = & \mathcal{F}_i(\mathcal{M}^{i}_{\mathcal{QM}^{+}[\psi]}[e + \gamma_1], \ldots, \mathcal{M}^{i}_{\mathcal{QM}^{+}[\psi]}[e + \gamma_n])
\end{array}
\tag{6.1}
$$

where $\epsilon = \{\epsilon_1, ...\epsilon_{dim}\}$ is the set of the vectors in the natural basis of $\mathbb{N}^{dim}$. Moreover, $\beta_\alpha * \epsilon_i$ is the scalar product which returns the component of the vector β_α along the main space direction indicated by the vector ϵ_i.

Informally, the transformation moves the application point only on the first dimension. With respect to the resulting relative shape, two situations are possible:

I The first coordinate of all the shape elements is strictly positive

II At most the first coordinate of only one element is equal. Moreover the element is the result of a translation of the application point only along the first axis.

We reported the transformed shape of a set of stencils in Figure 6.6. The darker element in the shape represents the application point or equivalently the centre of the reference system used to describe the relative shape.

Both Laplace and Jacobi fall into the second of the described configurations; there is at most one element whose first coordinate is equal to zero. In the other configuration is the nine point stencil; indeed all elements feature a first coordinate that is strictly positive. This difference will be extremely important in the proof of the following Theorem.

Theorem 6.2.1 (**In-Situ Computation with** $\mathcal{QM}-transformations$**).** *Let ψ be a generic $\mathcal{HUA}$ stencil. One step of $\mathcal{QM}^+[\psi]$ can be implemented at the concurrent level by exploiting an in-situ computation without copy overhead.*

Proof. Let the spatial structure of ψ be defined as follows:

$$\mathcal{M}_\psi = \frac{\mathbb{Z}}{m_1} \times \ldots \times \frac{\mathbb{Z}}{m_n}$$

For the implementation of the working domain at the concurrent level, we consider the following matrix:

$$J[m_1 + q_1^+][m_2] \ldots [m_n]$$

We suppose that, at the beginning of the step, the working domain values are mapped onto the first $m_1 - 1$ rows of the J matrix.

The first operation we consider is a copy operation which, exploiting some buffers, breaks the toroidal functional dependencies. In other words we break the functional dependencies, if any, between elements on the top of the matrix and those on the bottom. This copy operation is mandatory to target an *in-situ* computation and cannot be avoided.

It is worth mentioning that the number of copied elements depends on the stencil volume and is negligible compared to copying the entire working domain.

To prove the Theorem, we claim that the visit pattern schematized in Figure 6.8 respects the semantics of the $\mathcal{QM}^+[\psi]$ stencil computation.

Before proceeding, it is important to notice that the reference system of C-like matrices (the one used in sequential aspects of the concurrent language) features the first dimension with an opposite orientation to the shape reference system. Consider the property of the relative shape of featuring elements with a first component that is strictly non negative. In terms of the C-like matrices, the properties assert that the first index of each element is smaller than the index of the application point.

Because we are targeting an *in-situ* computation, the matrix J will store both input and output values for the computation of a step. Hence, the computation inside the for-all loop obviously features loop-carried dependencies with respect to the outermost loop controlled by the variable x_1.

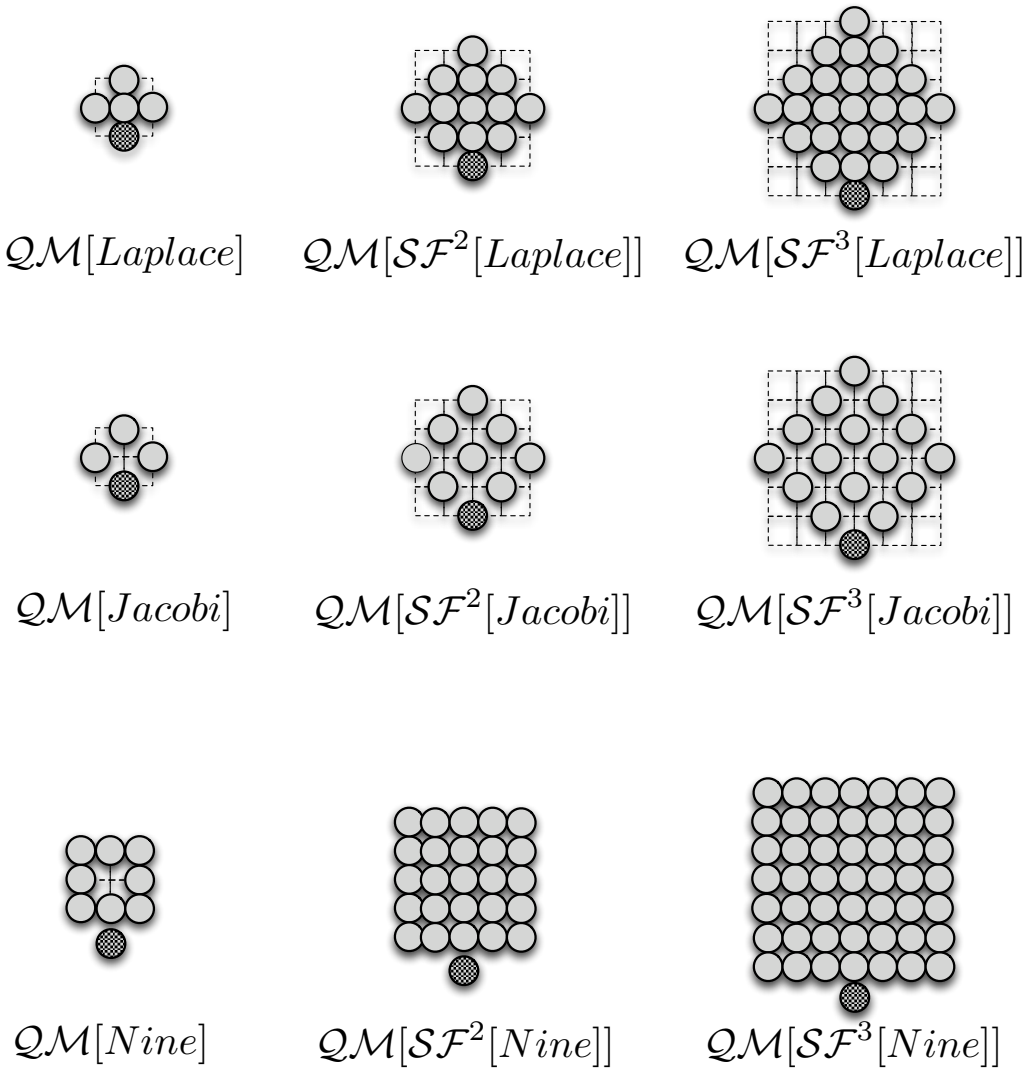


Figure 6.6: Graphical representations of shapes and and application points of a set of stencils transformed by $\mathcal{QM}-transformations$.

```
for(x_1 = m_1 − 1; x_1 > −1; x_1 − −){                          1
  forall((x_2, ..., x_n)|(x_1, x_2, ..., x_n) ∈ J)             2
  {                                                             3
    compute QM^+[ψ] with (x_1 + q_1^+, x_2, ..., x_n)           4
     as application point                                       5
  }                                                             6
```

Figure 6.7: Visit pattern forced by the *in-situ* computation

We have to demonstrate that the order of read and write operations which is forced by the specific pattern of visit respects the semantics of the program. For the sake of simplicity, we consider first the case $q_1^+ = q_{limit}^+ + 1$.

The selected visit pattern forces a decreasing order of visits for the first component. In other words, given two matrix indices $(\alpha_1, \ldots, \alpha_n)$ and $(\beta_1, \ldots, \beta_n)$, if $\alpha_1 > \beta_1$ then the first index is visited before the second.

Recalling that the visit pattern selects the application points for the computation, we analyze the relation between shape and visit pattern.

By definition, $\mathcal{QM}^+[\psi]$, in the case of $q_1^+ = q_{limit}^+ + 1$, features a relative shape whose first component is strictly positive. In terms of the C-like reference system, the application point features a first component that is higher than the component of the shape elements. Because only the shape elements are targeted for read operations, a matrix element is visited (i.e. considered as application point) only when there is no other element that requires its value.

To summarize, all the functional dependencies are oriented on indices featuring a strict lower value in the first component. A bottom-up visit along the first dimension guarantees the program semantics. This proves the Theorem for the case $q_1^+ = q_{limit}^+ + 1$.

The other case, $q_1^+ = q_{limit}^+$, is just an optimization for those stencils which, like Jacobi and Laplace, feature only one element of the shape with the lower first component. The element has moreover the characteristics of lying on the axis of the first dimension.

In these cases, it is possible to avoid the addition of one unit because no data dependency can arise between elements of $\mathcal{QM}^+[\psi]$ which feature the same value in the first component. □

We can therefore also claim the following Theorem.

Theorem 6.2.2 (Memory Requirement with Positive $\mathcal{QM}$–*transformations*). *Let ψ be a generic $\mathcal{HUA}$ stencil. One step of $\mathcal{QM}^+[\psi]$ can be implemented at the concurrent level by exploiting memory of size equal to*

$$|\mathcal{M}_{\mathcal{QM}^+[\psi]}| * d(\mathcal{D}_{Jacoibi})$$

plus some memory for buffer support, which according to Working Hypothesis 3.2 can be considered to be negligible.

Proof. The proof comes directly from the implementation of $\mathcal{QM}^{+}[\psi]$ proposed in the proof of the previous Theorem. □

6.2.2 Negative $\mathcal{QM}-transformation$

We have seen that positive $\mathcal{QM}-transformations$ can be used to implement a single step, carrying out an *in-situ* computation and without introducing a copy overhead of the complete working domain.

Nevertheless, positive $\mathcal{QM}-transformations$ cannot be applied to two consecutive steps. Indeed we recall that in the proof of Theorem 6.2.3, we assumed that the working domain elements are mapped at the beginning of the step onto the upper part of the matrix. After one step the domain values are stored in the lower part of the matrix.

We need some mechanisms to move the elements back to the top of the matrix. Or better, we can exploit the same technique that we used with the positive $\mathcal{Q}-transformations$. We therefore define an opposite transformation to the positive one.

Definition 6.2.2 (**Negative** $\mathcal{QM}-transformation$)**.** Let ψ be a $\mathcal{HUA}$ stencil. A negative $\mathcal{QM}-transformation$ is a $\mathcal{Q}-transformation$ that transforms ψ into the $\mathcal{HUA}$ stencil $\mathcal{QM}^{-}[\psi]$ which is equivalent to the original one, except for the step model that is defined as follows:

$$
\begin{array}{rcl}
\forall e \in \mathcal{M} \overset{\mathcal{QM}^{-}[\psi]}{\rightarrow} & \left(\mathcal{F}^{\psi}, \mathcal{S}^{\mathcal{Q}^{-}[\psi]}\right) \\
q_m^- = & < q_1^-, 0 \ldots, 0 > \\
q_{limit}^- = & -max\left\{\beta_\alpha * \epsilon_1 \forall \beta_\alpha \in \mathcal{R}^{\psi}\right\} \\
q_1^- = & \begin{cases} q_{limit} \textbf{if } (\beta_\alpha * \epsilon_1 = -q_{limit}^-) \Rightarrow (\forall j > 1 \ \beta_\alpha * \epsilon_j = 0) \\ \\ q_{limit}^- - 1 \text{ in the other cases} \end{cases} \\
\mathcal{R}^{\mathcal{QM}^{-}[\psi]} = & \mathcal{R}^{\psi} + q_m^- \\
\mathcal{S}^{\mathcal{QM}^{-}[\psi]} = & e + \mathcal{R}^{\mathcal{Q}^{-}} \\
\overset{\Downarrow}{=} & e + \left\{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_\alpha = \beta_\alpha + q^-\right\} \\
\mathcal{M}^{i+1}_{\mathcal{QM}^{-}[\psi]}[e] = & \mathcal{F}_i(\mathcal{M}^{i}_{\mathcal{QM}^{-}[\psi]}[e+\gamma_1], \ldots, \mathcal{M}^{i}_{\mathcal{QM}^{-}[\psi]}[e+\gamma_n])
\end{array}
\tag{6.2}
$$

where $\epsilon = \{\epsilon_1, ...\epsilon_{dim}\}$ is the set of vectors in the natural basis of $\mathbb{N}^{dim}$. Moreover, $\beta_\alpha * \epsilon_i$ is the scalar product which returns the component of the vector β_α along the main space direction indicated by the vector ϵ_i.

As for the positive $\mathcal{QM}-transformations$, we can claim the following Theorem for the negative ones.

Theorem 6.2.3 (**In-Situ Computation with Negative** $\mathcal{QM}$**–***transformations*)**.** *Let ψ be a generic $\mathcal{HUA}$ stencil. One step of $\mathcal{QM}^{-}[\psi]$ can be implemented at the concurrent level by carrying out an in-situ computation without copy overhead.*

Proof. Let the spatial structure of ψ be defined as follows:

$$\mathcal{M}_{\psi} = \frac{\mathbb{Z}}{m_1} \times \ldots \times \frac{\mathbb{Z}}{m_n}$$

For the implementation of the working domain at the concurrent level we consider the following matrix:

$$J[m_1 - q_1^{+}][m_2] \ldots [m_n]$$

We suppose that, at the beginning of the step, the working domain values are mapped onto the last $m_1 - 1$ rows of the J matrix.

The first operation we consider is a copy which, by using some buffers, breaks the toroidal functional dependencies. In other words we break the functional dependencies, if any, of the elements on the top of the matrix with those on the bottom. This copy operation is mandatory to target an *in-situ* computation and cannot be avoided.

It is worth mentioning that the number of copied elements depends on the stencil volume and is negligible compared to copying the entire working domain.

To prove the Theorem, we claim that the visit pattern schematized in Figure 6.8 respects the semantics of the $\mathcal{QM}^{+}[\psi]$ stencil computation.

```
for(x_1 = 0; x_1 < m_1; x_1 ++){                                  1
  forall((x_2, ..., x_n)|(x_1, x_2, ..., x_n) ∈ J)                2
  {                                                               3
    compute QM^-[ψ] with (x_1, x_2, ..., x_n)                     4
     as application point                                         5
  }                                                               6
```

Figure 6.8: Visit pattern forced by the *in-situ* computation.

From this point on, the demonstration is completely symmetric with respect to the one given for positive $\mathcal{QM}-transformations$.

It is easy to see that at the end of one step computation the values of the working domain are now stored in the upper part of the matrix, i.e. in the first m_1 rows. □

We can therefore conclude with the following Theorem.

Theorem 6.2.4 (**Memory Requirement with Positive** $\mathcal{QM}$**–***transformations*)**.** *Let ψ be a generic $\mathcal{HUA}$ stencil. A generic number of steps ψ can be implemented at the concurrent level by interleaving positive and negative $\mathcal{QM}$–transformations: $\mathcal{QM}^{+}[\psi]$ on odd steps and $\mathcal{QM}^{-}[\psi]$ on even ones.*

The resulting implementation has a memory requirement equal to

$$|\mathcal{M}_{\mathcal{QM}^{+}[\psi]}| * d(\mathcal{D}_{Jacoibi})$$

plus some memory for buffer support which according to Working Hypothesis 3.2 can be considered to be negligible. Finally, the implementation does not imply the complete copying of all working domain elements.

Proof. The proof comes directly from the union of all the other Theorem proofs. □

6.2.3 Performance Tests

We tested $\mathcal{QM}-transformations$ for different stencils, architectures and also combining $\mathcal{SF}-transformations$. The studied stencils are Jacobi, Laplace and Nine plus all their transformations with the step fusion method up to a level of four.

We exploited three different architectures whose characteristics are given in the following list.

I Intel Pentium III CPU - M @ 800MHz cache size 512 KB

II Intel Pentium 4 CPU @ 2.00GHz cache size 512 KB

III Intel Xeon CPU E5420 @ 2.50GHz cache size 6144 KB

For each of the three stencil we provide a chart plotting the following functions:

$$\mathcal{G}^{\mathcal{SF}[\psi]}(i) = \frac{i * T^{\psi}}{T^{\mathcal{SF}^{i}[\psi]}}$$

$$\mathcal{G}^{\mathcal{QM}[\mathcal{SF}[\psi]]}(i) = \frac{i * T^{\psi}}{T^{\mathcal{QM}[\mathcal{SF}^{i}[\psi]]}}$$

The variable T^{ψ} is the mean time to compute once the entire working domain, using the stencil ψ. For the test, we obviously considered the computation of a working domain whose size would fit completely into the main memory, but not into the cache levels.

The first function defines the time gain of the stencil $\mathcal{SF}^{i}[\psi]$ relative to the naive implementation $\mathcal{SF}^{1}[\psi] = \psi$. This is the same parameter we presented in the chart of the previous Chapter to study $\mathcal{SF}-transformations$.

The second function defines the time gain of the whole stencil $\mathcal{QM}[\mathcal{SF}^{i}[\psi]]$, again relative to the naive implementation $\mathcal{SF}^{1}[\psi] = \psi$. Therefore, the two time gains are relative to the same reference implementation.

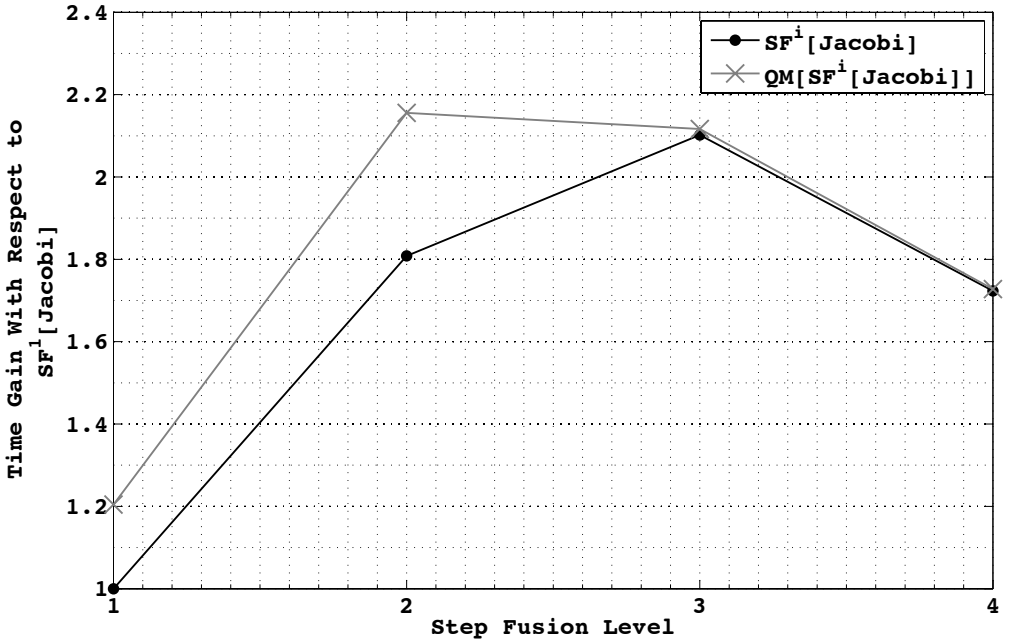
From all the charts reported in Figures 6.11, 6.12, 6.13 and 6.14, a well-defined trend is evident. In all cases, the exploitation of $\mathcal{QM}-transformations$ improves the performance of the computation for low levels of step fusion. More precisely, the best performance configuration, which in the charts is represented by the point with the higher time gain, is always associated with a stencil that has been transformed with $\mathcal{QM}-transformations$.

Therefore the transformation both reduces memory requirements and improves performance. The performance benefit comes from the impact of the algorithm on cache hierarchy; more precisely on write operations.

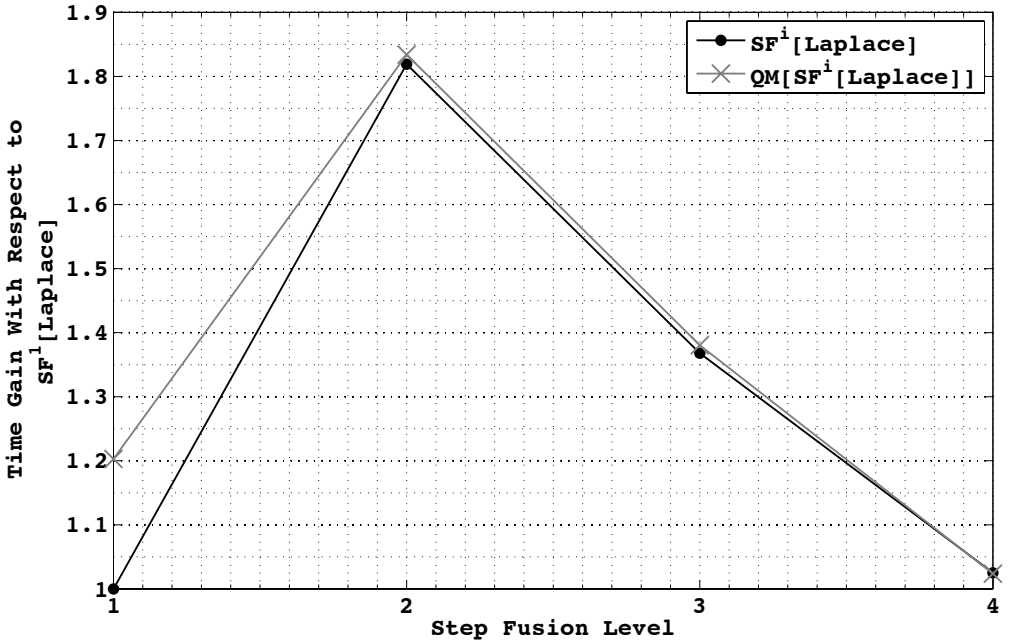
In write allocate architectures, a write miss will necessitate the allocation of a cache line. Therefore, before the execution can proceed, the contents of the line must be read from main memory. In the case of a stencil computation, this is a waste of time because the entire line will be rewritten anyway.

A possible way of avoiding this situation is to use cache initialization and cache bypass instructions. According to Datta et al. [15], the use of this instruction can increase the performance by 50%.

Without exploiting cache bypass instructions, $\mathcal{QM}-transformations$ provide an implementation of the Laplace and Jacobi stencils that avoids the overhead of unnecessary cache misses. Indeed, the results are stored in cache lines that have been previously read.

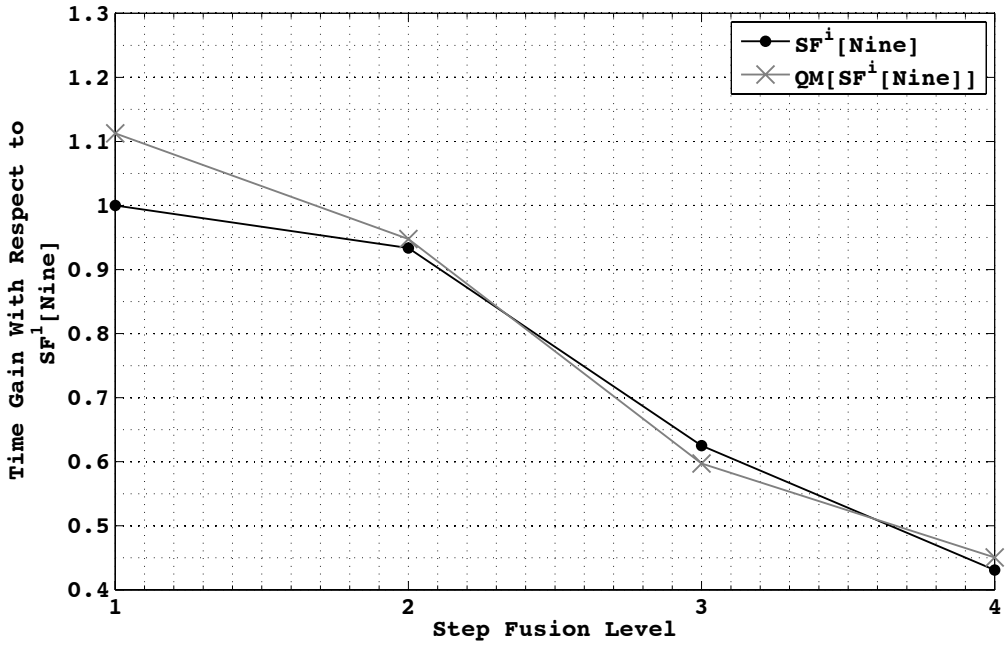


(a)



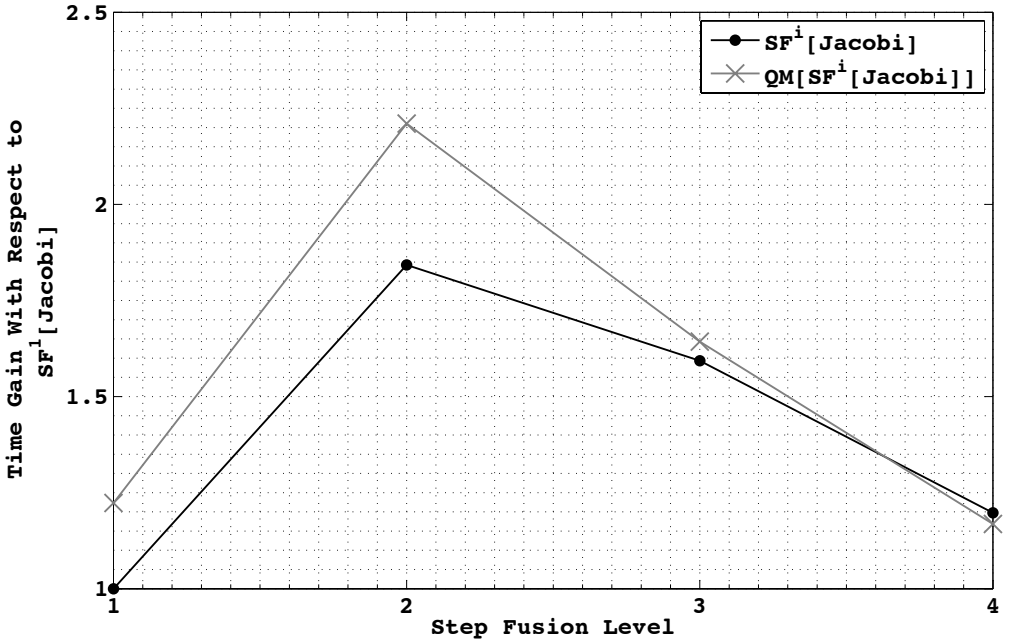
(b)

Figure 6.9: Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB

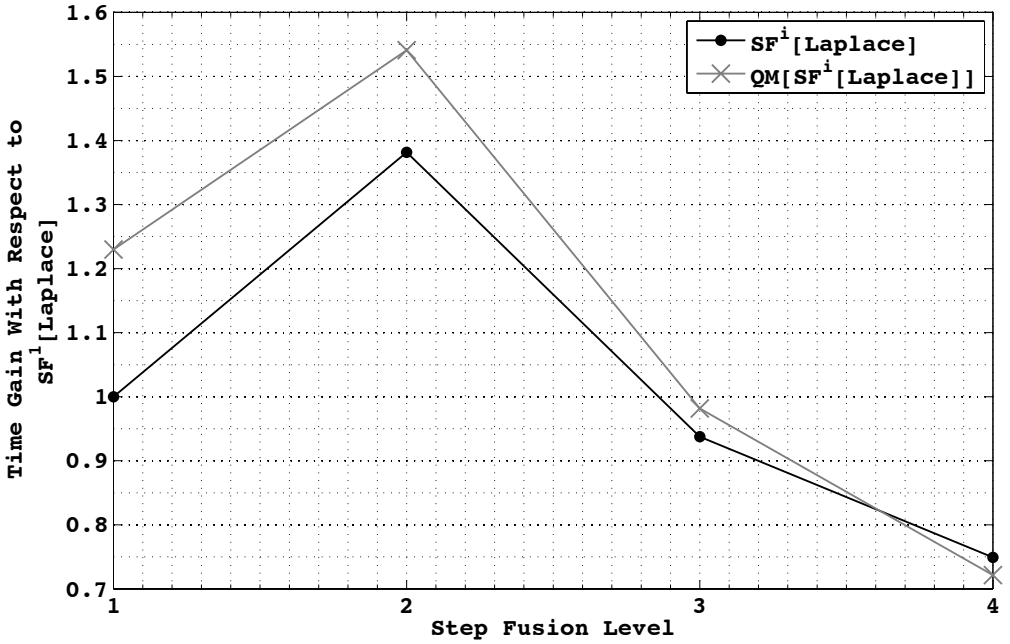


(a)

Figure 6.10: Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB

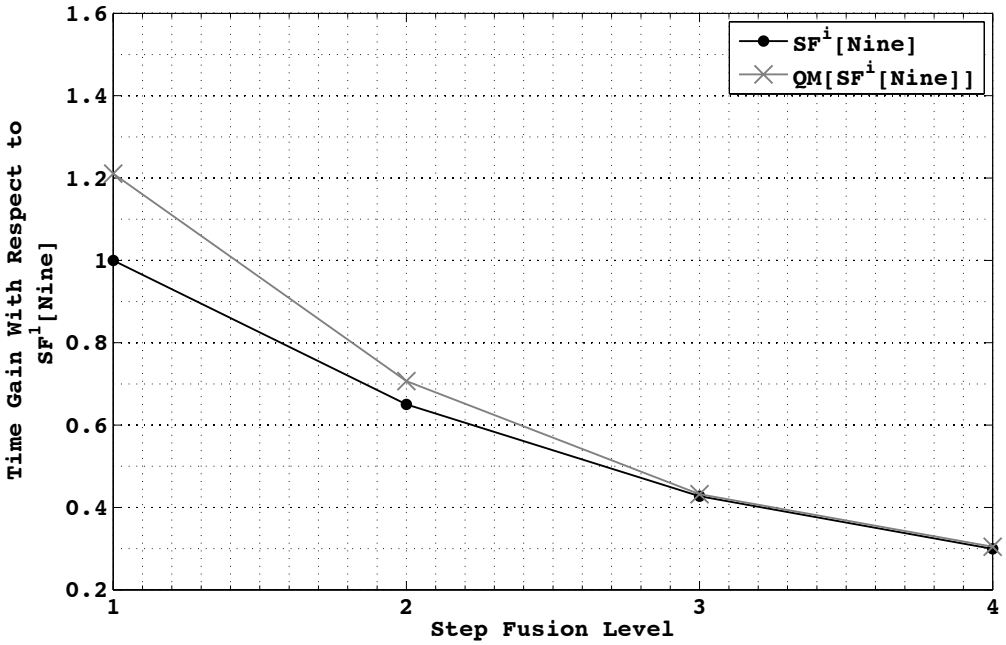


(a)



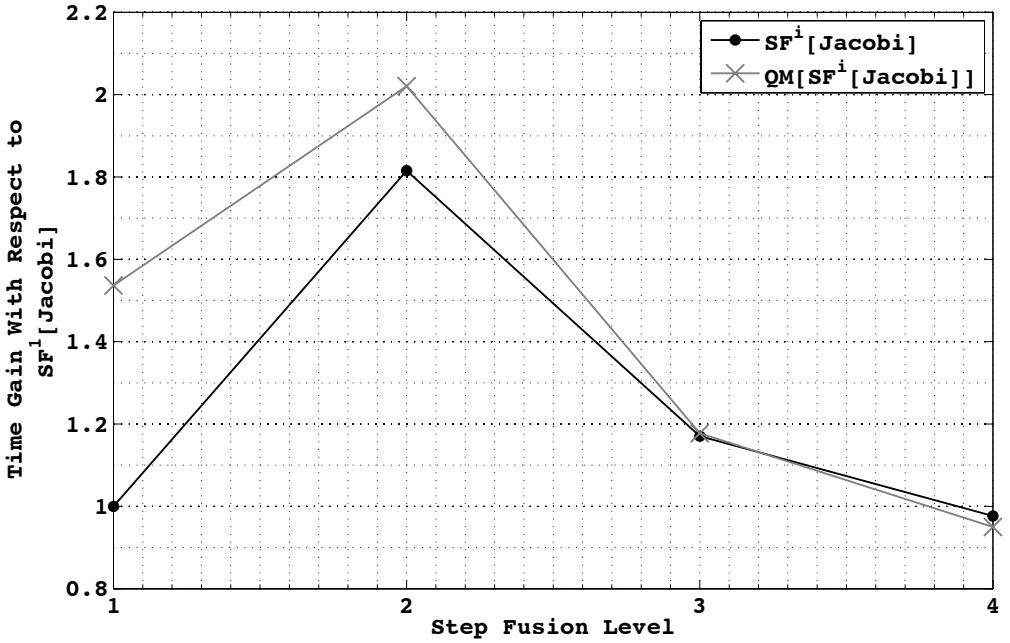
(b)

Figure 6.11: Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB

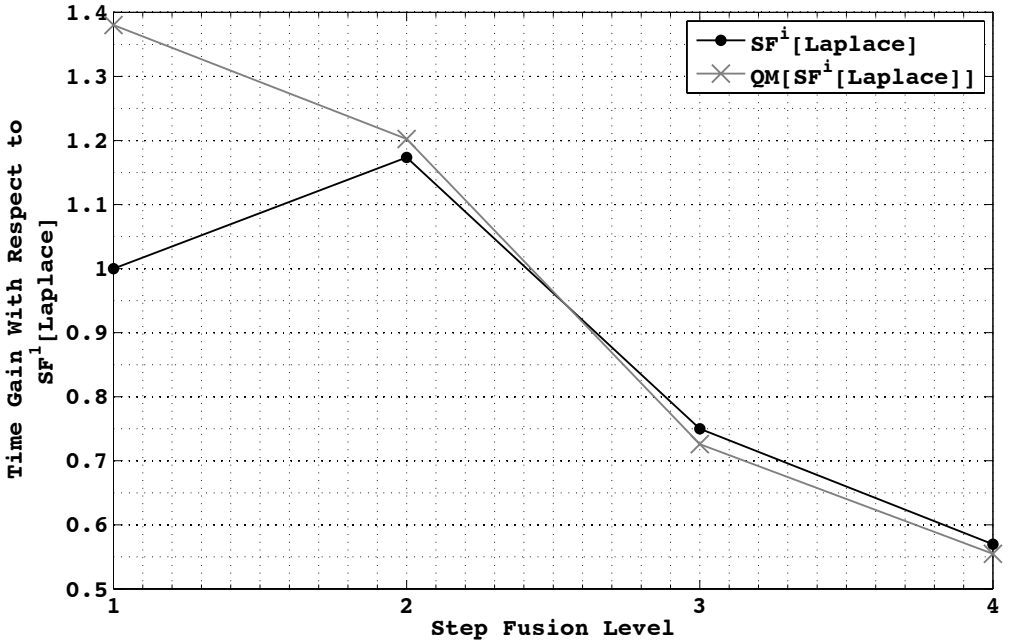


(a)

Figure 6.12: Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB

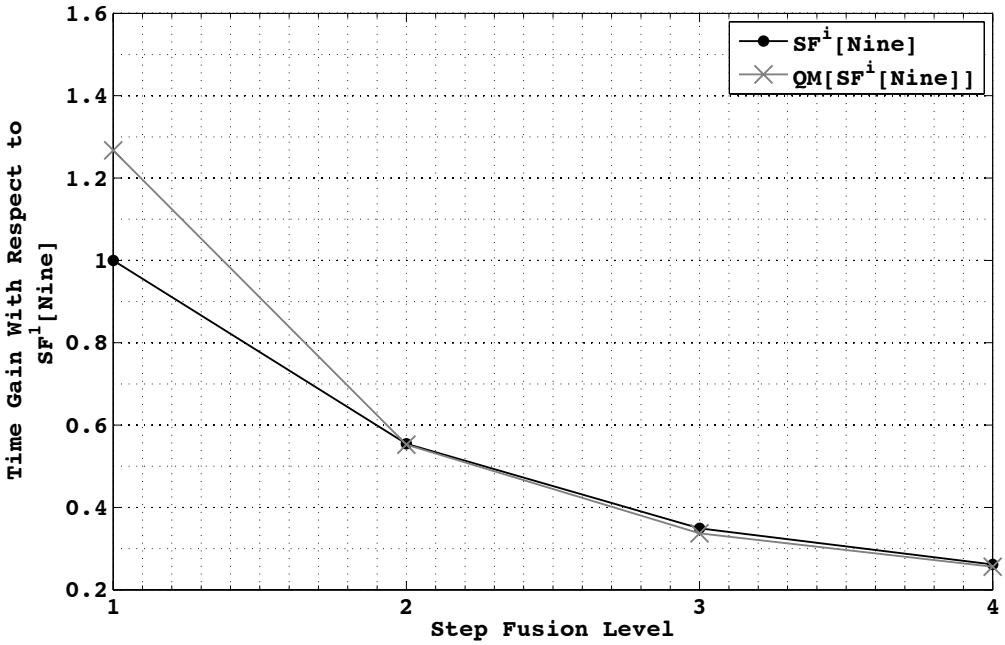


(a)



(b)

Figure 6.13: Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB



(a)

Figure 6.14: Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB

Chapter 7

Conclusions

In the previous Chapters, we presented a set of innovative transformations for stencil-based applications.

- $\mathcal{Q}$–$transformations$ provide optimizations for communication overhead. With the transformations presented, the number of communications required to implement a generic stencil, defined over an n-dimensional space, can be reduced to n, relative to $2 * n$, which is the best result provided by solutions cited in the literature.

  Experimental tests on clusters and multi-core architectures prove that $\mathcal{Q}-transformations$ provide better performance than other implementations, especially when targeting fine grain parallelization. In worst case tests, where communications with all neighbours are required, the reduction of the communication overhead has been quantified with respect to a "naive" implementation in terms of a time gain of up to 4.5.

- $\mathcal{QSF}-transformations$ are a formalization and notable extension of these techniques which exploit replication of data to reduce communications. One notable useful side effect of studying $\mathcal{QSF}-transformations$ is the definition of strategy for memory hierarchy management in stencil computational kernels.

  The technique, which is based on a revisit of classic loop fusion optimizations, provides a time gain of up to 2.1, in a sequential environment.

- $\mathcal{QM}-transformations$ result in optimizations that target the reduction of memory constraints. This reduction is obtained without performance loss; the type of memory accesses defined by $\mathcal{QM}-transformations$ rather provide notable performance benefits.

  In experimental results, we prove that $\mathcal{QM}-transformations$ almost halve the memory constraints and in some cases provide a time gain of up to 2.2.

All the transformations presented are the result of a new optimization theory centered both on a modification of the well know owner-computes rule and on base but powerful properties of toroidal spaces.

In contrast to the classic approach, we relay on a two phase optimization technique. In a first phase we focus on program transformations that are relaxed-safe: the optimized program and the original one produce, from the same input data structures, the same output values, which however can have a different spatial organization in the output data structures. In a second phase we exploit smart mechanisms in order to convert the output back into the original one.

The two phase optimization theory, which can be modelled as a optimized case of the owner-sotre rule, along with a structured approach to studying stencil-based application gives us the basis for defining a structured stencil model which is exploited to introduce and formally prove all the previous transformations.

Formally, the relaxed-safe property is an extension of the classical theory. Indeed, each transformation that is safe in the classical meaning can be obviously classified as relaxed-safe; but the other way round, the implication is not valid.

From the previous assertions, we claim that we can merge classical optimization techniques with the newly presented techniques in order to obtain other relaxed transformations that can used in the first phase of our optimization theory. We consider this perspective an interesting and promising subject for future works

Appendix A

$\mathcal{M}$ammu$\mathcal{T}$

Abstract

Structured parallel programming is a parallel software development methodology that aims to deliver programmability, portability and interoperability, along with scalability and performance. To achieve these goals, it is important to define both a suitable set of high level parallel constructs and a communication language whose mechanisms provide very high performance and low overhead, for the efficient implementation of parallel construct runtime, and a clear cost model that allows for parallel construct composition optimization.

We describe our experience in defining abstract and concrete communication protocols optimized for structured parallel programming on single chip multi-core architectures. We implement and test our mechanisms on the IBM Cell BE chip multi-core. We detail a comprehensive cost model of communications, which is a requirement for supporting automatic optimization in a structured parallel framework, and we report the achieved performance. Our implementation reaches best possible bandwidth and latency figures on this architecture: measured performance numbers are extremely close to actual hardware limits.

The Chapter is structured as follow. Section A.1 describes our structured parallel programming framework.

Section A.2 introduces the semantics and syntax of the communication language $\mathcal{LC}$; the section also briefly compares some important characteristics of $\mathcal{LC}$ and MPI.

Section A.3 and A.4 describes an abstract protocol and platform-independent optimization for $\mathcal{LC}$ channels.

Section A.5 details the actual implementation of communication support on the Cell architecture.

Finally Section A.6 reports achieved performance, comparing it with MPI for Cell implementation and with hardware *DMA* transfer mechanisms. This section characterizes also the communication cost model.

Contents

NOWADAYS, EVERY CPU DESIGN is based on integrating multiple homogeneous or heterogeneous cores inside a single chip. This technology shift, from accelerating single core performance to integrating multiple simpler cores, originates from the inability to scale application performance using only hardware improvements and instruction level parallelism. Current chip production technology suffers from severe physical limitations, which can be summarized as power, frequency and memory walls: we have a hard limit on usable power in integrated circuits, diminishing returns from deeper pipelines that allow faster clock speed and the access speed of DRAM memory is severely limited.

CPU designers attack these power, frequency and memory issues by requiring programmers to explicitly express thread level parallelism and exploit this parallel behaviour on multiple (more or less independent) cores. Software developers are forced to use techniques similar to those developed for traditional high performance computing (*HPC*) architectures. What has changed from the past is the packaging: these new architectures are now integrated inside a single chip. In this scenario, as it has always been in the *HPC* field, the burden of achieving software performance shifts from hardware designers to software developers: in order to gain any advantage from modern *CMP*, every software component must be coded with explicit parallelism.

A new software development methodology is clearly fundamental in order to be able to provide desirable characteristics such as programmability, portability and interoperability, along with performance and scalability. *Structured parallel programming* is an approach aiming to deliver these goals by "restricting" the parallel program structure to a set of parametric and configurable parallel constructs. This approach drives programmability by defining a methodology that guides programmers through the design of a parallel application, while hiding low level architectural aspects and allowing application portability on different architectures. On the other hand, structured parallel programming allows for the definition and implementation of tools that automatically perform static and run-time optimization, both for performance and for scalability. These tools are strictly based on the definition and validation of a cost model for all mechanisms used in the language runtime implementation.

We describe our experience with developing a set of low level mechanisms, based on the message passing paradigm, which constitute an intermediate "language" for inter-process communication and are able to support the development of a runtime for structured parallel programming constructs. The Appendix details two achievements:

I We define a set of communication mechanisms featuring a clear cost model for inter-process interactions.

II We show how to exploit the knowledge about structured parallel constructs in order to implement optimized strategies for communication mechanisms

We refer to the set of communication mechanisms by the terms *communication support* while we name *communication language* the language that manages them.

We therefore present a generic description of a communication support for structured parallel programming and its implementation on the *IBM Cell BE* CMP architecture. Performance tests show that bandwidth and latency obtained for one-to-one communication between processes on different cores are close to actual hardware performance limits. We show how our implementation achieves the lowest possible overhead and we compare our results with the *IBM Cell BE MPI* implementation showing that our solution provides one order of magnitude improvement.

Finally we define a cost model of the communication mechanisms, which enable the implementation of parallel constructs, defining their specific cost model, and study algorithms for automatic optimization.

A.1 Structured Parallel Programming

The key idea of structured parallel programming (SPP) is to describe a parallel application using only a specific set of parallel mechanisms, or parallel constructs (PCs), and combining them. PCs are based on concepts like replicating or partitioning a function or data, e.g. farm, map and stencil data parallelism, divide&conquer, etc. [14, 19, 30, 7, 52, 5]

This methodology provides a straightforward strategy for analyzing and parallelizing applications, thanks to the separation of parallel and sequential aspects and to the hiding of all low level details of communication and synchronization.

SPP methodology alone is not a complete solution for *CMP* programming or parallel programming in general. While PCs allow for an easy description of a parallel application, the development, configuration and tuning of the corresponding program is not a simple task. An efficient implementation of a parallel construct in an architecture requires an in-depth knowledge of important low level aspects of the specific target architecture, as well as good skills for the design and management of synchronization and communication.

Moreover a structured parallel application must take into account configurations that better overlap communication and computation. A load balancing strategy is mandatory when managing irregular problems. Communications between processes should be vectorized (i.e. grouped) to obtain a lower overhead. Finally the individual sequential processes, which set up the parallel application, must be mapped and scheduled on physical resources. Algorithms that find exact solutions for most of these problems have been proved to be NP-hard.

We study SPP with the purpose of defining algorithms to find reasonable approximate solutions for the previous NP-hard problems, exploiting information that can be extracted from an in-depth knowledge of the PC semantics and a PC cost model for a target architecture.

Our approach to parallel programming consists in defining a complete framework

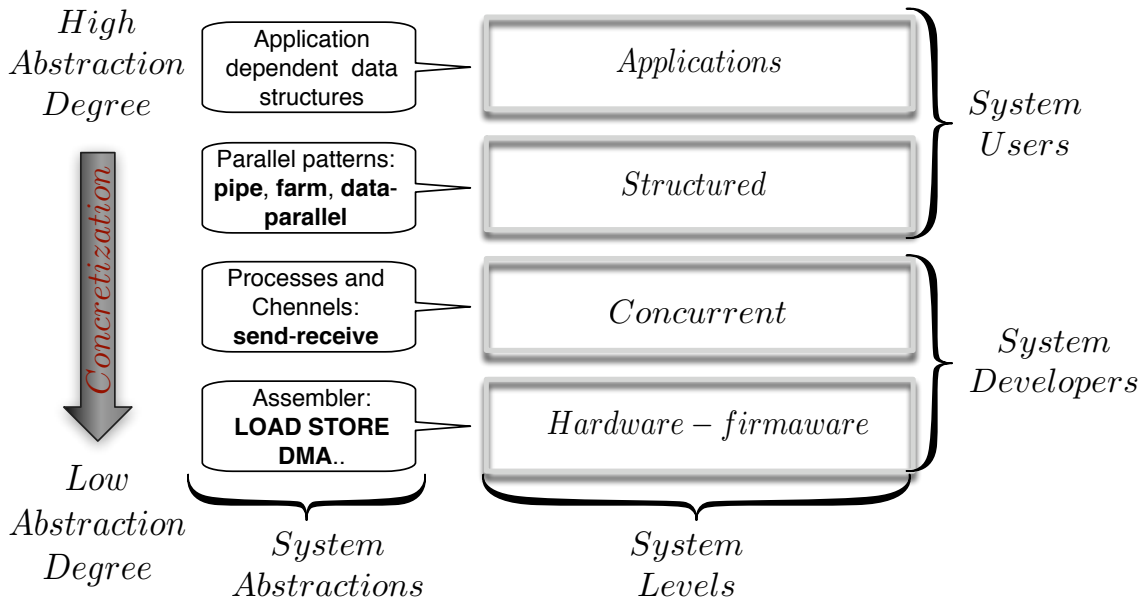


Figure A.1: System structure

based on SPP methodology. The framework includes a system for implementing a set of PCs and a set of algorithms to manage both the optimization and configuration required to run an efficient parallel program.

We structure our framework in four hierarchical levels (see Figure A.1); from top to bottom we have *application, structured, concurrent* and *hardware-firmware* levels. Each level defines, and at the same time is based on, a programming language featuring a particular degree of abstraction. We recall that we use the term "level" and "language" in an interchangeable way. The architecture we present now is a generalization of the one that we introduced in Chapter 3.

The *structured language* provides to application developers high level constructs to design and implement parallel applications according to the SPP methodology. Developers declare and aggregate PCs, composing them to achieve the desired application behaviour. Programmers can therefore rely on a high degree of abstraction since all low level details are completely hidden inside the individual PC implementation.

The *concurrent* level, which represents the *communication support*, is characterized by a language that expresses parallel processing as a set of interacting sequential computational kernels. In general, threads can interact using either a shared memory or a message passing mechanism; in our framework we use a message passing paradigm. This language is designed in such a way as to facilitate the implementation, modelling and configuring of a system for SPP.

Following the hierarchical relation between levels, all the parallel constructs of the structured language are implemented by strategies defined in the concurrent language; for example a farm construct can be described as the process (*master*) which

uses some communication mechanism to distribute tasks to a set of *worker* processes and then collects the results. Since we require the strategy (i.e. implementation) of PCs to be portable and independent of the specific target architecture, we use the concurrent language to completely mask the characteristics of the underlying concrete architecture.

The lowest level of the system is the hardware-firmware, where the basic support for managing sequential processes and communication is implemented. At this level, a parallel application is described by a set of operations on hardware registers of the target architecture.

Every system level must be characterized by a cost model, which is an essential element for optimization and configuration purposes. In this Appendix we focus on the two bottom layers: concurrent and hardware-firmware layers.

Several different message passing libraries are available in industry and academia, for example MPI is the most used communication support in HPC. Most of these solutions provide enough mechanisms to define PCs. Our team has extensive experience in developing SPP environments on a wide range of architectures: from clusters to grids. In this environment, we used standard *communication supports* for many reasons, such as portability, compatibility and saving of development time.

In a *CMP* environment, standard mechanisms are too heavy weight, since they were designed for classical MPP architectures, so we decided to distill the required mechanism into a simple and low overhead interface.

A.2 $\mathcal{LC}$ Language Semantics and Syntax

$\mathcal{LC}$ is the language we defined for the concurrent level; it is based on a message passing paradigm and describes abstract processes communicating according to a local environment model. $\mathcal{LC}$ was born as a tailored minimal formalism of the *CSP* model of Hoare [22].

An $\mathcal{LC}$ parallel program statically declares a set of processes, that can also be parametrically described in terms of some identifiers. Processes do not share any data structures: channels, statically defined in the program (no channel can be created at run time), are the only mechanism that describe interaction between entities.

A channel is unidirectional and typed: it represents a queue where a set of producer processes insert messages and a single consumer process extracts data. The type of channel and the type of messages in its queue must match. Channels can implement either symmetric communications, when only one producer is defined, or asymmetric communications.

An important characteristic of $\mathcal{LC}$ channels is the degree of asynchrony, which is a static integer parameter defining the maximum number of non-blocking send operations that can be performed when no receive operation is invoked. When the parameter is equal to zero, the channel is synchronous by definition. The degree of asynchrony is defined statically and has to be guaranteed by the implementation of the language for the lifetime of the $\mathcal{LC}$ program.

It is important to observe that semantically asynchronous MPI communications are equivalent to $\mathcal{LC}$ channels with an infinite degree of asynchrony. MPI programmers don't need to define a degree of asynchrony but, as a consequence, they prevent the MPI support from exploiting important information for optimization as we will show later.

Finally $\mathcal{LC}$ includes an alternative guarded command, similar to the command in *CSP*, to manage non-deterministic aspects; this is a key relevant construct for dynamic load balancing aspects.

A.2.1 $\mathcal{LC}$ Channel API

We developed the first version of $\mathcal{LC}$ implementation on the $IBM\ Cell\ BE$ architecture, implementing a *C++* library called **MammuT** (*cell* **M***ulticore* **A***rchitecture co*-**MMU**-*nication suppor*-**T**). We used the *C++* language rather than *C* to exploit an object-oriented interface and to benefit from template mechanisms both for static type checking and compile time optimization.

The **MammuT** library includes a set of methods that describe, also in a parametric way, the graph of an application and therefore how to define processes, channels and bindings. In this appendix, we do not describe the declaration of processes and channels: we focus only on the channel construct interface, implementation and performance.

In the **MammuT** library, a channel is a C++ template object with the following signature:

```
template <int Role ,type msg_type,
      int Async_degree, int Window_length>
Spe_channels <Role, msg_type,
      Async_degree, Window_length>
```

The `Role` template parameter sets the channel as incoming or outgoing, the `Type` parameter establishes the type of messages that can be sent over the channel and `Async_degree` determines the degree of asynchrony. `Window_length`, which is detailed in section A.4, defines a logical lifetime for receive messages on a specific channel.

The runtime interface of a MammuT channel includes the following methods:

- `int transfer_id send(msg_type * msg)`: used to send messages over a channel. The method returns an identifier for the transfer.
- `void wait(int transfer_id)`: used to wait for the completion of the send operation identified by the transfer_id
- `msg_type * receive()`: used to retrieve a message from an incoming channel.

A.2.2 MPI and $\mathcal{LC}$

We are not studying $\mathcal{LC}$ as a language to replace MPI. We investigate $\mathcal{LC}$ because its expressiveness is sufficient to express PCs and it presents static features that can be exploited to achieve high performance.

While comparing the MPI interaction model with $\mathcal{LC}$, it is evident that the latter requires several pieces of static information about the communications: in $\mathcal{LC}$ you must specify both end points of every channel, the message type and the fixed degree of asynchrony. These pieces of static information are relevant to introducing optimizations and to targeting more efficient communication protocols. The same information is required also when using MPI communication, but it is available only at run time.

Along with performance optimization, a complete knowledge of the communications between processes, and especially the characteristics of a fixed degree of asynchrony is enough to characterize the $\mathcal{LC}$ program memory requirement for asynchronous communications. This is a sensitive aspect especially while targeting an architecture with memory limitations such as the SPE cores inside the Cell BE multi-core.

The MPI standard states that a program is "safe" only if it relies on synchronous communication [50]. This limitation is due to the inability to guarantee that the

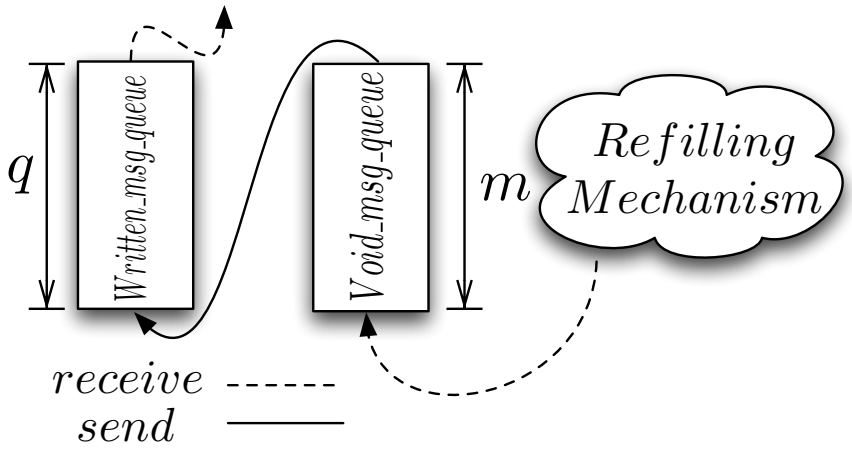
memory required for buffering asynchronous communication can be allocated at runtime; if memory is not available the parallel application will deadlock.

In the MPI model, the *Buffer Allocation Problem* (BAP), which is the problem of determining the minimum number of buffers required to ensure a deadlock-free execution, has been proved to be NP-hard independently from the actual communication buffer management strategy [11]. Moreover the *Buffer Sufficiency Problem* (BSP), which is the problem of deciding if a given buffer assignment is sufficient for deadlock-free executions, is still intractable when exploiting a receiver buffer allocation strategy[11], which is the only mechanism which provides zero copy communication.

In contrast to MPI, $\mathcal{LC}$ solves the problem of buffer allocation by design. If it is not possible to allocate at start-up enough space for the required buffers, whose dimension is known at compile time thanks to $\mathcal{LC}$'s static semantics, the program cannot be executed. Thus, once the program starts, programmers don't need to take into consideration the deadlock situation due to buffer allocation.

$\mathcal{LC}$'s features make it possible to introduce several optimizations (detailed in Section A.3 and A.4), but we are not proposing that $\mathcal{LC}$ should be used instead of MPI: $\mathcal{LC}$'s semantics has been defined to support the definition of parallel skeletons only. We use the $\mathcal{LC}$ language for the concurrent level instead of MPI, in order to exploit the performance gain that can be extracted from the more rigid semantics.

A.3 $\mathcal{LC}$ Channel Abstract Protocol



(a)

```
send(ch_type the_msg){
   pop(Void_msg_queue, new_msg);
   write(new_msg, the_msg);
   push(Written_msg_queue, new_msg);
}

ch_type receive( ){
   pop(Written_msg_queue, new_msg);
   get_void_msg(Refilling_mechanism,void_msg);
   push(Void_msg_queue, new_entry);
   return new_msg;
}
```

(b)

Figure A.2: Pseudo-code and data structure used by the $\mathcal{LC}$ channel abstract protocol

In this Section, we detail the abstract implementation of a one-to-one $\mathcal{LC}$ channel, independently of the specific architecture. This abstract description helps us to define the important characteristics of our channel support and to show some optimizations and the featurs of the $\mathcal{LC}$ channel that enable them. This abstract description is an introduction to the concrete implementation on the Cell architecture.

$\mathcal{LC}$ one-to-one channels are typed and asynchronous: a send operation can terminate before the corresponding receive is called. To implement this behaviour, it is mandatory to provide some memory locations where messages can be written. The number of messages already inserted into the channel plus the number of predefined memory locations used to store future messages must be equal to or greater than the degree of asynchrony k of the channel. We call this the $k - property$.

A one-to-one communication channel is normally represented as a $FIFO$ queue containing the messages inserted into it. In our representation, as shown in figure A.2(a), we consider a second queue to model the preallocated messages, as required by the $k - property$. We name the first queue $Written\_msg\_queue$, the second $Void\_msg\_queue$ and we define q and m their respective lengths.

Both queues are shared between sender and receiver processes and are managed using only push and pop operations. The protocols followed by sender and receiver are described in pseudo-code in figure A.2(b), which shows that the behaviour of the two are somehow dual.

The sender first pops from the $Void\_msg\_queue$ a reference to a memory location where the message can be written and, after writing the data, it pushes the reference back into the $Written\_msg\_queue$. The receiver starts popping from the $Written\_msg\_queue$ a reference to a sent message and then, before returning the message, it defines and inserts a new memory location into the $Void\_msg\_queue$.

As previously highlighted by the $k - property$, the sum of q, the number of messages that have been sent but not yet received, and m, the number of memory locations predefined to store future messages, has to be always equal or greater than k.

This protocol preserves the $k - property$. During the initialization phase, we fill the $Void\_msg\_queue$ with k references and we empty the $Written\_msg\_queue$; as a consequence, we have $q + m = 0 + k = k$. From this point on, both send and receive procedures maintain the sum of q and m equal to k. At each call, the send procedure decreases m, by extracting an element from the $Void\_msg\_queue$, but increases q, by inserting an element into the $Written\_msg\_queue$. The reverse happens when handling a receive operation, which decreases q and increases m.

It is important to highlight two features of the protocol: first it requires that pop operations are blocking only when the queue is empty; and secondly it does not require any queue length check for push operations: push operations are always allowed.

Finally, sender push and pop operations are performed on a different queue with respect to receiver push and pop operations. These queues are then characterized by one producer and one consumer: their implementation can use a lock-free algorithm, even without a mechanism such as fetch-and-add or compare-and-swap, as shown by Lamport [37, 18]. This property is relevant when targeting high performance especially on the Cell, where atomic operations for the *SPE* core exhibit high latency.

It is important to underline that most of the nice characteristics of the protocol we have presented are strictly linked to the static type of channel and to the fixed degree of asynchrony: the degree of asynchrony gives information about the minimum number of preallocated messages required to guarantee the $k - property$, while the type of information specifies the memory required for every message.

The get_void_msg method wraps the possible mechanism for defining memory location where future messages can be written. Because of the utility of these mechanisms, we refer to them as *refilling* mechanisms; several refilling solutions are

possible and we will present and analyze them in the following section.

A.4 Channel Abstract Optimization

A.4.1 Static Refilling: the *w_protocol*

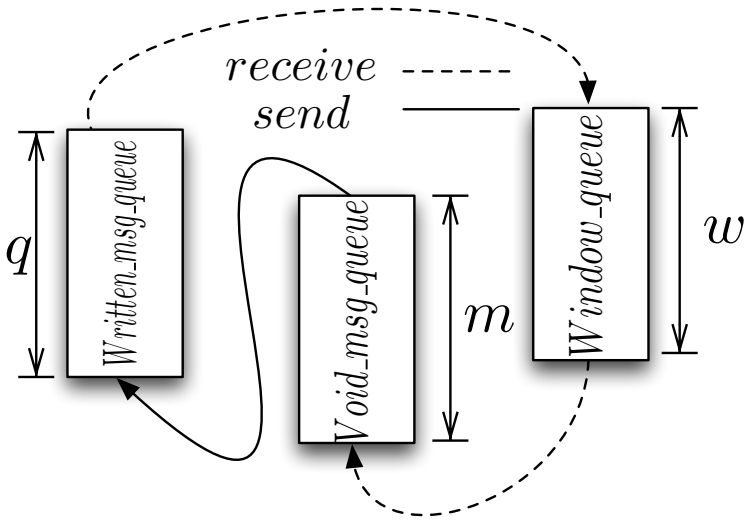


Figure A.3: Data structures used by the $\mathcal{LC}$ abstract protocol *w_protocol*

```
ch_type receive( ){
   pop(Written_msg_queue, new_msg);
   push(Window_queue, new_msg);
   pop(Window_queue,new_entry);
   push(Void_msg_queue, new_entry);
   return new_msg;
}
```

Figure A.4: Pseudo-code of the $\mathcal{LC}$ abstract protocol *w_protocol*

We begin the abstract channel performance optimization section with the definition of an efficient *refilling* mechanism.

A first possible solution is based on dynamic memory allocation: at each receive, a new memory chunk is allocated and its reference is inserted into the $Void\_msg\_queue$. This *refilling* mechanism implies an allocation overhead at each receive and requires $\mathcal{LC}$ programmers to manage the de-allocation of received messages. This solution also implies that the parallel program implementation will not be deadlock-free.

A second potential solution requires programmers to pass as a parameter of the receive method a reference to a new memory location for receiving the message. With respect to the first solution, this *refilling* mechanism avoids the dynamic allocation overhead, but both solutions manage message references that are not defined statically.

A third possible implementation is based on statically defined references. This last solution is obviously not compatible with a zero copy strategy: on the receiver side, messages have to be copied from the static buffer to another memory location. The mechanism therefore reduces data traffic but increases the receive latency with message copy overhead.

We selected an alternative solution that can be implemented with static message references, to avoid additional communication between sender and receiver, but allows for a zero-copy feature that avoids the unnecessary copying of messages. The mechanism is based on the introduction of additional constraints on the semantics of the received messages.

In most of the PCs, such as farm, pipe and also often in data parallel constructs, the behaviour of the threads implementing the pattern can be schematized as "activations". During each activation few messages are received and the data are used as parameters by the computational kernel; after the computation has been completed, a set of response messages is sent. The messages received can also be used to modify the state of the process but at the end of the processing step they are not needed any more and can be de-allocated.

To take advantage of this characteristic behaviour of parallel constructs, we introduce a *refilling* mechanism based on buffer entry reuse, which introduces a usage time restriction on received messages.

We start from the abstract representation of the channel protocol in Figure A.2(b) and we consider a third queue, called $Window\_msg\_queue$, which is used only by the sender (see Figure A.3). We define w as the number of elements in this queue and we require it to be a static constant defined by programmers. At channel initialization, we allocate $k + w$ messages and we push k message references into $Void\_msg\_queue$, w and $Window\_msg\_queue$.

The new protocol on the receiver side (Figure A.4) pops the reference of the message to be received and pushes it directly into the $Window\_msg\_queue$. The receiver *refilling* mechanism simply pops a reference from the $Window\_msg\_queue$ and pushes it into the $Void\_msg\_queue$; the $k - property$ is still guaranteed.

Following the path of a single message reference for a set of send calls, it is evident that it is periodically reused to write new messages. Therefore a received message is characterized by a logical lifetime, which depends on the $\mathcal{LC}$ channel parameter w. A received message can be used, without introducing incoherence, so long as its reference remains in the the $Window\_msg\_queue$.

From the programmers' perspective, a received message can be accessed for a limited time. When the $w + 1$ receive function call is done, the message data will not be valid any more since its reference is extracted from the $Window\_msg\_queue$ and pushed into the $Void\_msg\_queue$.

Programmers can statically set the value of w depending on the parallel construct behaviour in accessing incoming messages. The higher the value of w, the longer the message lifetime is. In most cases, where only a single message is used to send parameters to a computational kernel, a value of w equal to one is enough

to guarantee zero copy. It is relevant to underline that the parameter w does not modify the semantics of the degree of asynchrony.

We call the abstract protocol optimized with the *refilling* mechanism the *w-protocol*.

A.4.2 The K_plus_one Optimization

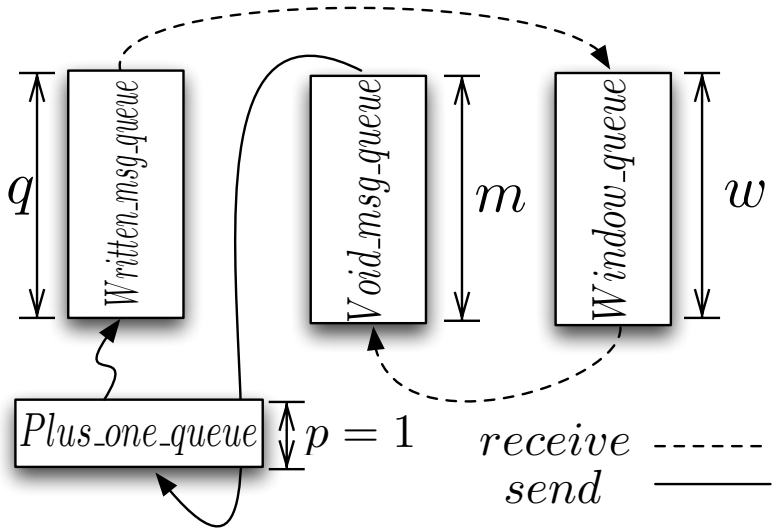


Figure A.5: Data structures used by the K_plus_one optimization

```
send(ch_type the_msg){
   pop(Plus_one_queue, new_msg);
   write(new_msg, the_msg);
   push(Written_msg_queue, new_msg);
   pop(Void_msg_queue, next_msg);
   push(Plus_one_queue, new_msg);
}
```

Figure A.6: Pseudo-code of the K_plus_one optimization

We now describe an optimization strategy which allows for better overlapping between communication and computation of the send procedure. Although this optimization concerns only the sender side, and therefore is completely independent of the specific *refilling* mechanism, we present it in the context of the *w-protocol*. This new optimization uses another queue called $Plus\_one\_queue$ (see Figure A.5). The queue is managed only on the sender side and the number of references it stores, represented by the variable p, is constantly equal to one.

In the new send procedure (figure A.6), the sender first pops from $Plus\_one\_queue$ a message reference, which is used to write the message to be sent, and pushes it

back into the $Written\_msg\_queue$. Finally, to keep p equal to one, the send support pops a reference from the $Void\_msg\_queue$ and pushes it directly into the $Plus\_one\_queue$.

As in the other version of the protocol, at initialization k references are defined and inserted into the $Void\_msg\_queue$, but another reference is defined and inserted into the $Plus\_one\_queue$. The total number of references requested at channel setup is equal to $k + w + 1$; for this reason we refer to this optimization by the name k_plus_one.

The $k - property$ is still guaranteed by the length of $Written\_msg\_queue$ and $Void\_msg\_queue$: in other words, because the patterns of access to the shared queues have not changed, we still have $q + m = k$.

The benefits of introducing $Plus\_one\_queue$ are evident from comparing the send pseudo-codes in fig A.4 and A.6. The improved performance comes from the fact that the message copy in the k_plus_one version is computed before the pop operation on $Void\_msg\_queue$. The pop on the shared queue is the only operation that can potentially require a wait. The pop on $Plus\_one\_queue$ never waits since, between one send and the next, p is always equal to one: the queue is never empty.

Thus, a possible stall, due to the blocking semantics of the pop operation, is postponed in time, allowing the computation of some useful work. When a blocking pop returns, the send procedure in the k_plus_one version must still copy the message.

Moreover, neglecting the latency of the pop operation on the private $Plus\_one\_queue$, the write is the first operation to be computed. If the architecture supports asynchronous writes, the operation overhead is overlapped with the rest of the protocol.

A.5 Concrete Implementation on the Cell

A.5.1 The Cell Architecture

The *Cell* chip contains nine heterogeneous computational elements linked through a component called the *Element Interconnection Bus* (*EIB*). This element is composed of four 16 byte wide data rings, each allowing up to three concurrent data transfers (if there is no physical overlap). The *EIB* acts like a connection-orientated network and supports the transfer of blocks of up to 128 bytes. See [4] for an in-depth presentation of the *EIB*.

The other Cell elements are: the Power Processing Element (PPE), eight Synergistic Processor Elements (SPEs), one Memory Interface Controller (MIC) and two I/O units. The PPE is an "in-order" 64 bit IBM Power 4 processor with two threads. Each SPE features a Synergistic Processor Unit (SPU), a memory flow controller (MFC) and a very fast 256 Kbyte $SRAM$ memory (called local store - *LS*) with a 6 clock latency; the SPU is a $RISC$-style CPU optimized for vector operations.

The *MFC* acts as a memory management unit and a *DMA* engine; *DMA* is the only way to move data between the local store and other SPE or main memory. The *DMA* engine schedules operations asynchronously and in an unordered way but some DMA commands (fence and barrier) are provided to force an ordering.

A.5.2 Channel Implementation on Cell

In this section we describe the implementation of $\mathcal{LC}$ channels in the Cell architecture based on the *w-protocol* and the *k_plus_one* design.

The *w-protocol* allows us to statically allocate a vector of messages (r_msg_vect) of length $k + w + 1$; this is the only buffering support that will be exploited. To guarantee zero copy, the buffer is allocated to the *LS* of the receiver process. A vector of references (s_ref_vect), each pointing to a corresponding location in r_msg_vect, is instantiated on the sender side. A reference can be used to write into the corresponding message through a *DMA* operation.

As the receiver message buffers are statically allocated at initialization time and its entries are reused during the program run, $s\_ack\_bit\_vect$ is initialized once and never changed: no information exchange from receiver to sender is required to keep message references up-to-date, therefore saving both latency and bandwidth.

To avoid synchronizations through lock mechanisms, which in the $IBM\ Cell\ BE$ architecture exhibits high latency and cannot be completely delegated in an asynchronous way to the *MFC*, we follow an alternative implementation of the Lamport algorithm [37].

The Lamport technique works for one consumer/one producer queues and it is based on the management, in a non atomic way, of a top reference, modified only by the receiver, and a bottom reference, modified only by the sender. The read of a

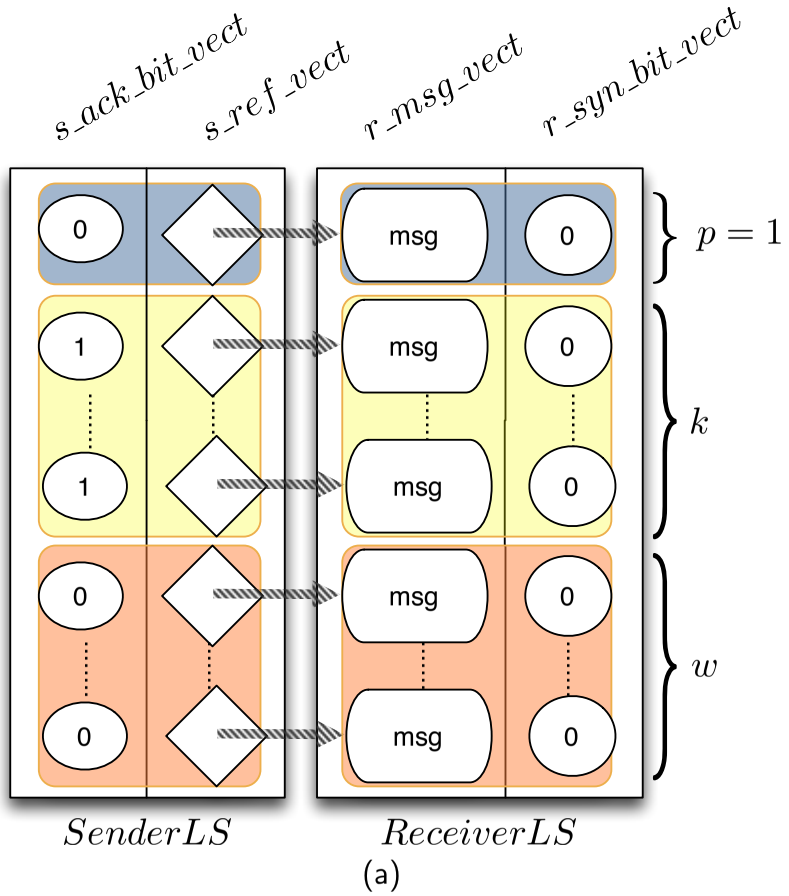


Figure A.7: Cell $\mathcal{LC}$ channel data structures (A.7(a)) and its concrete protocol compared with the abstract one (A.8(a)).

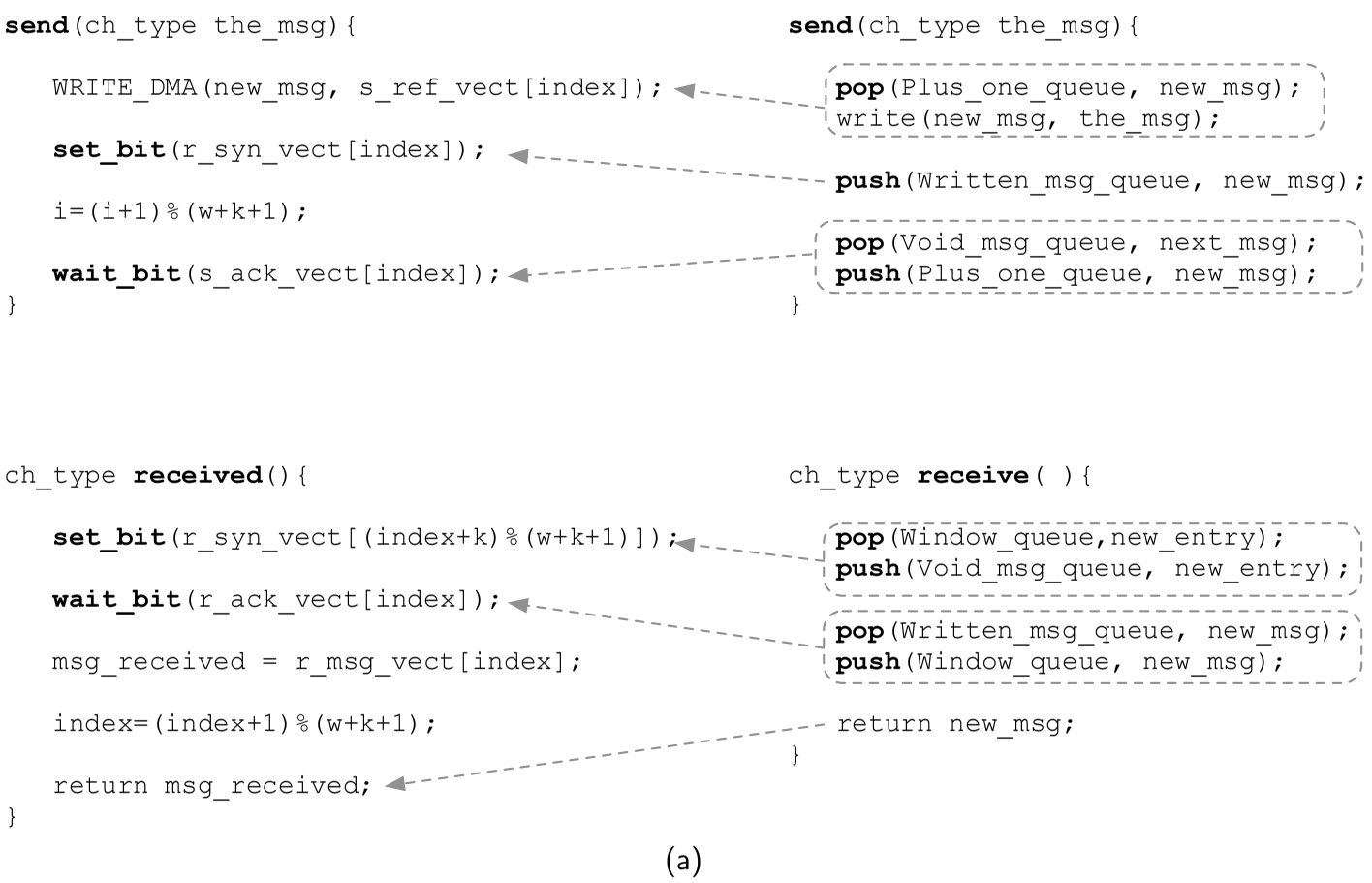


Figure A.8: Cell $\mathcal{LC}$ channel data structures (A.7(a)) and its concrete protocol compared with the abstract one (A.8(a)).

non up-to-date value, which can result in a stall due to full or empty queue status, is followed by retries. This solution fits well the *IBM Cell BE* environment where *SPE* are mono-programmed and can efficiently use busy waiting.

We follow a slightly different approach, which is semantically equivalent to Lamport. Each pair, formed by an element of *r_msg_vect* and the corresponding one of *s_ref_vect*, is associated with two bits of information. Bits are logically structured, as presented in Figure A.7(a), in two vectors: *s_ack_bit_vect*, allocated to sender *LS*, and *r_syn_bit_vect*, to receiver *LS*.

The semantics of a bit is straightforward: a bit of *s_ack_bit_vect* is equal to one when its message belongs to *Void_msg_queue*. Symmetrically, when *r_syn_bit_vect* is equal to one, the corresponding message belongs to the *Written_msg_queue*.

We need two operations on bits in these vectors: *wait_bit* and *set_bit*. The first operation, which will be used only on local data, blocks until the bit value is equal to one and resets it to zero: the operation stalls until the message is in the *Void_msg_queue* and automatically extracts it. The *set_bit*, which is used only on remote bits, sets a bit value to one; the operation is used to insert a message into the *Void_msg_queue*.

At initialization, we specify the bit values as shown in figure A.7(a) which describes the mapping between the data structure entries and the queue of the abstract protocol. Since at program startup no message is present in the *Written_msg_queue*, all *r_syn_bit_vect* bits are set to zero.

On the sender side, we associate the first entry with the *Plus_one_queue*, thus its associated bit is set to zero. The following k entries are mapped into the *Void_msg_queue* and their bit is set to one. Finally the rest of the vector is mapped onto the *Window_msg_queue* and all the bits are set to zero.

Both sender and receiver work on the data structures in a circular way, incrementing a local reference (called *index*) which points at the vector entries that must be used. At initialization, the value of *index* is equal to zero for both sender and receiver: for the sender it points to the entry associated with the *Plus_one_queue*. For the receiver, the variable points to the first position that is going to be inserted into *Written_msg_queue*. The complete protocol to manage all data structures is shown in Figure A.8(a), where the *WRITE_DMA* represents a *DMA* transfer.

From an architectural perspective, *set_bit* is an operation called by the program on the *SPU* but its execution is delegated to the *MFC*. A nice feature on the sender side is that both the first two operations, *WRITE_DMA* and *set_bit*, are delegated to the *MFC* in an asynchronous way: the sender process does not wait for their completion. The remaining part of the send protocol is overlapped with the actual MFC transfer of the message and the associated *set_bit*.

The protocol described in the previous section can be implemented on the Cell architecture using different low level mechanisms for *set_bit* and *wait_bit*. The Cell provides two main techniques for synchronizing *SPE*s.

The first option is to use point-to-point communication over a special *MFC* regis-

ter, called the signal register. We call the channel implementation that exploits this kind of mechanism signal-based implementation (*DMAS*). This solution leverages both the blocking semantics of signal registers (when reading a zero value) and its *OR* writing semantics: the write operation of a value on the signal register stores in the register the result of an *or* operation between the old value and the new one. The *or* behavior, which can be set when loading a program on an *SPE*, is useful for introducing an optimized computation based on bit masks. The *SPE-SPE* communications through the signal register are performed by *DMA* transfer. To distinguish these *DMA* communications from the other transfers, we refer to them as signals.

The second synchronization technique is equivalent to the previous one but it performs polling on local storage locations and not on special registers. We can define two different channel implementations for this strategy depending on the number of transfers used: a single transfer (*DMA1*) or a double transfer (*DMA2*).

A.5.3 Signal-based Implementation

In this solution, data movement is accomplished using a *DMA* operation, while the *set_bit* primitive is implemented by sending a signal to the remote *SPE*. The signal must be sent using a *fence* instruction to ensure that the presence bit is set to one only after the first *DMA* completes. Thanks to the *MFC* unit, the *SPU* can simply queue the *DMA* transfer, which continues asynchronously, and then check for the empty queue event. The overall latency of this implementation is composed of the *DMA* transfer and signal latencies. In fact, since the queue empty test is performed locally, no overhead is added over the physical communication.

A.5.4 DMA1 Implementation

Analyzing the signal-based implementation, it is clear that adding one *EIB* transfer only for synchronization purposes adds the overhead of establishing a new connection over the *EIB* ring for moving the minimum message size (16 byte). To avoid this overhead we implemented the *set_bit* in a different manner: this solution merges both the message and synchronization in the same *EIB* connection.

We consider the message as a structure, named "*enlarged*" message: it contains the original messages plus a single byte that acts as a bit of *r_syn_bit_vect*, thus all buffer entries on the receiver side are "enlarged messages". The *wait_bit*, instead of polling on a signal register, polls into the local storage the synchronization byte of the message that has to be received. The main requirement for this solution to work correctly is that the synchronization byte must be written after the message data: we must know the details of the *EIB* transfer policy.

In the current Cell architecture, there is no guarantee on the order in which a set of *DMA* operations are executed except when inserting a fence or barrier instruction; moreover when a single transfer is larger than 128 bytes it is split into

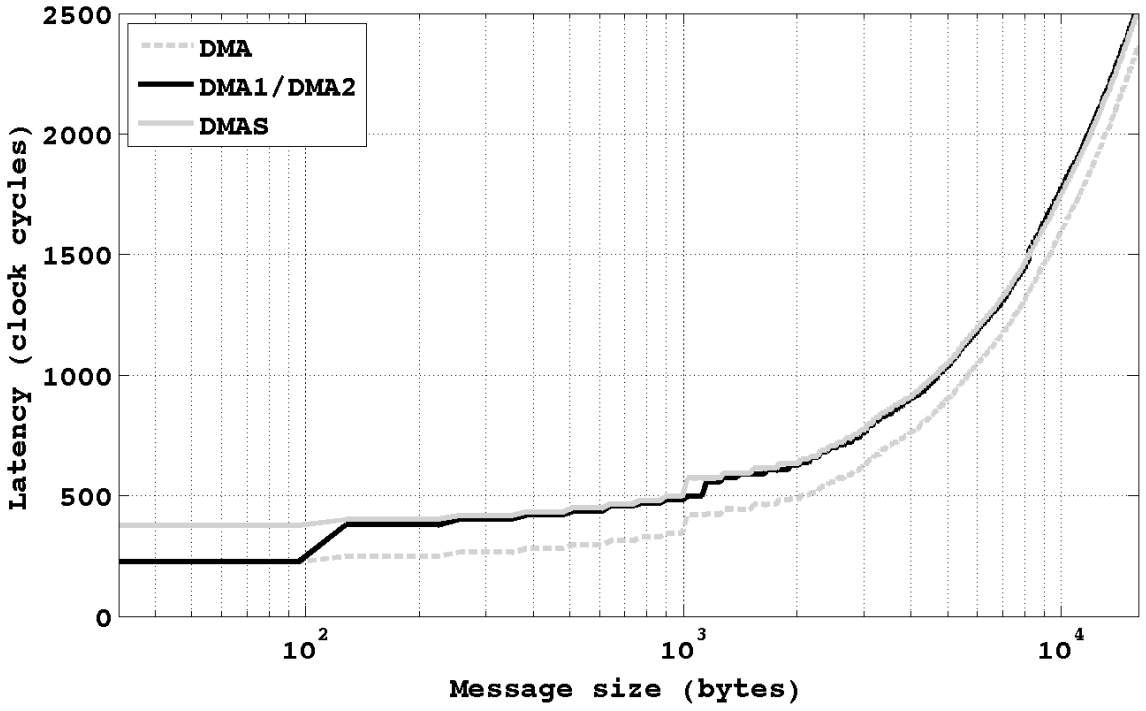
more transfers which are executed out of order. Unfortunately there is no single mechanism to guarantee that the synchronization data byte is written last.

Our *DMA1* implementation works only for special messages where the enlarged message location is 128 byte aligned and its size is up to 128 bytes. When this holds, the *MFC* transfers the data in order, starting from the lowest address. Inserting the synchronization byte after the message data will ensure the correctness of the operation.

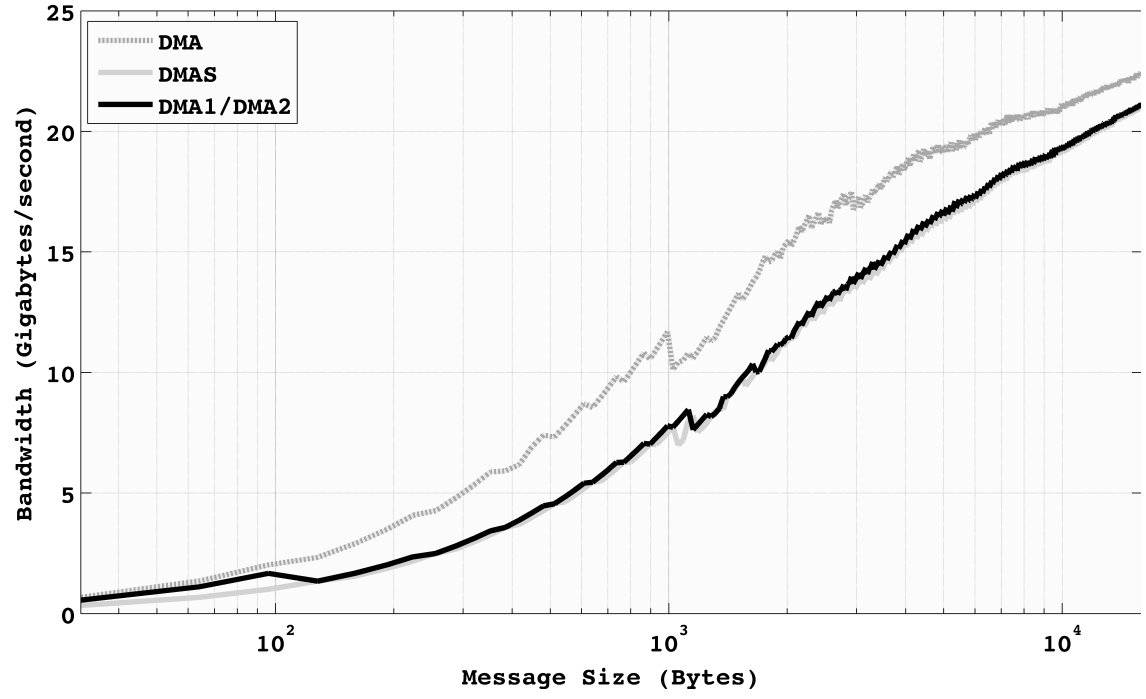
A.5.5 DMA2 Implementation

The last implementation strategy we tested is still based only on *DMA* operations but is able to transfer messages larger than 128 bytes. The basic idea is still to exploit the compile time knowledge of type size and structure the physical data layout to transfer the information efficiently over the EIB.

The weakness of the signal implementation is that delivering a signal requires a connection to be established over the EIB to transfer the 32 bit signal word. For example, if we have a message containing 184 bytes, the signal-based implementation will create 3 separate connections over the *EIB*: the first moves the initial 128 bytes, the second moves the next 56 bytes and the last delivers the 4 byte signal word. The *DMA2* solution is still based on the idea of merging message and synchronization data and achieves this by attaching the synchronization byte to the last "not full" *DMA* block. This requires messages to be 128 byte aligned and the implementation of the *set_bit* consists in an initial *DMA* transfer of the largest message chunk multiple of 128 bytes followed by a second *DMA* operation to transfer the remaining data plus a char for synchronization. Between the two transfer operations, a fence instruction is inserted to ensure that the synchronization data is written after the message content. The double *DMA* implementation exhibits a communication latency which corresponds to the physical latency of the two fenced *DMA* transfers.



(a)



(b)

Figure A.9: Comparison of latency and bandwidth performance of different $\mathcal{LC}$ channel implementations and of DMA transfer

A.6 The Cost Model

We evaluated our $\mathcal{LC}$ channel implementations on an $IBM\ QS21\ IBM\ Cell\ BE$ blade; the results are detailed in A.9(a) and A.9(b). All tests were run in the unloaded EIB scenario, therefore avoiding any kind of conflict on the interconnection structure. Since the *DMA1* and *DMA2* implementations are respectively developed for short and long messages,they are represented by the same curve tagged $DMA1/DMA2$.

The chart in Figure A.9(a) represents, for different message sizes, the latency of the channel implementations ($DMA1/DMA2$ and $DMAS$) and of the pure DMA (DMA).

The $DMA1$ implementation reaches optimal latency for short messages: equal to the latency of actual DMA transfers. This means that for messages smaller than 128 bytes, the overhead of sending an "*enlarged*" message instead of the pure one is negligible.

For longer messages, $DMA2$ and $DMAS$ are quite similar; they do not show relevant performance differences. Both of them feature an overhead, compared to the pure DMA, lower than 200 clock cycles.

We can assert that it is not possible to develop an $IBM\ Cell\ BE$ communication mechanism that reaches better performance. In each implementation we exploit, according to the $IBM\ Cell\ BE$ hardware and firmware architectural constraints, the minimum number of transfers to copy the message and advise about operation completion.

All three protocols follow a lock-free algorithm to manage channel shared queues. This means that we can completely avoid two negative characteristics of $IBM\ Cell\ BE$ atomic operation: they present high latency and they cannot be completely delegated to the *MFC*. With lock-free mechanisms, our implementations present a service time of about 140 clock cycles; this is the time a send call spends before returning. The measured value is lower than the lowest latency of an $\mathcal{LC}$ channel: in any configuration the protocol overhead is completely overlapped with the DMA communication required.

The chart in Figure A.9(b) shows how the performance bandwidth of the $\mathcal{LC}$ channels saturate at up to 85% of the maximum bandwidth featured by DMA transfer.

Comparing these results with the MPI performance reported in [28, 33, 53] we can see that our solutions achieve more than one order of magnitude better latency: for a 128 byte message, our $DMA1$ implementation uses just 250 clocks while the MPI implementation uses 6500 clocks. Our solutions also provide a much higher bandwidth: for example with 64 kbyte messages, $DMA2$ transfers data at almost 20 Gbps and Cell MPI runtime provides just 6 Gbps. The main impact on the performance difference is the fact that the strict semantics of $\mathcal{LC}$ allows us to pre-allocate buffers in the receiver local storage and never copy incoming data.

As previously mentioned, one of the most important aspects for developing a

framework for structured parallel programming is the availability of a detailed cost model for the communication; this is a fundamental element to predict program performance and to compare different possible optimizations and tunings. The cost model of our implementation as follows:

$$L_{com}(msg) = \begin{cases} (0.1363 * size_{msg} + 365)\ clocks\ cycles \\ \qquad \text{if } size_{msg} > 128\ bytes \\ \\ 250\ clocks\ cycles \\ \qquad \text{if } size_{msg} < 128\ bytes \end{cases} \tag{A.1}$$

The current limitation of this solution is the requirement to pre-allocate buffers on the receiver side which, given the size of SPE local store, can be problematic. We are currently investigating several solutions to this issue.

A.7 Conclusion and Future Works

We approached the issue of building a high performance and predictable support for handling data communication in a structured parallel programming framework for chip multi-core systems. We review the available communication libraries and conclude that they are too heavyweight to be used in chip multi-core systems, therefore we design the $\mathcal{LC}$ communication language, which is a subset of the *CSP* model. We then detail the abstract communication protocol and some abstract optimization strategies for the $\mathcal{LC}$ semantics.

We describe our concrete implementation of $MammuT$, a library for $\mathcal{LC}$, on the $IBM\ Cell\ BE$, where we exploit all the available features of the architecture to focus on achieving highest possible performance. Experimental results, obtained on an IBM QS21 Cell blade, show that both $MammuT$ latency and bandwidth are very similar to actual hardware and firmware performance limits. Moreover, our implementation adds minimal overhead and provides one order of magnitude improvement over the $IBM\ Cell\ BE\ MPI$ implementation. The results obtained derive from static information extractable from $\mathcal{LC}$ that allows the development of higher performance and better optimized communication protocols.

Bibliography

[1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computing*, 30(5):341–356, 1981.

[2] B. D. Acunto. *Computational Methods for DPE in Mechanics*. World scientific, 2004.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] T. W. Ainsworth and T. M. Pinkston. Characterizing the Cell EIB on-chip network. *IEEE Micro*, 27(5):6–14, 2007.

[5] M. Aldinucci, M. Coppola, M. Danelutto, and N. Tonellotto. High level grid programming with ASSIST. *Computational Methods in Science and Technology*, Jan 2006.

[6] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 112–125, New York, NY, USA, 1993. ACM.

[7] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SKIE: A heterogeneous environment for HPC applications. *Parallel Computing*, Jan 1999.

[8] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[9] U. Banerjee, S.-C. Chen, D. J. Kuck, and R. A. Towle. Time and parallel processor bounds for Fortran-like loops. *IEEE Transactions on Computing*, 28(9):660–670, 1979.

[10] Vincent Bouchitté, Pierre Boulet, Alain Darte, and Yves Robert. Evaluating array expressions on massively parallel machines with communication/computation overlap. *International Journal of Supercomputer Applications and High Performance Computing*, 9, 1995.

[11] A. Brodsky, J. B. Pedersen, and A. Wagner. On the complexity of buffer allocation in message passing systems. *Journal of Parallel and Distributed Computing*, 65(6):692–713, 2005.

[12] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, October, 1988.

[13] H. Casanova, A. Legrand, and Y. Robert. *Parallel Algorithms*. CRC Numerical Analysis and Scientific Computing Series. Chapman & Hall, 2008.

[14] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30, Jan 2004.

[15] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[16] Chris Ding and Yun He. A ghost cell expansion method for reducing communications in solving pde problems. pages 50–50, 2001.

[17] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, 1988.

[18] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 43–52, New York, NY, USA, 2008. ACM.

[19] D. Goswami. From design patterns to parallel architectural skeletons. *Journal of Parallel and Distributed Computing*, 62(4):669–695, Apr 2002.

[20] J. Guo, G. Bikshandi, B. B. Fraguela, and D. A. Padua. Writing productive stencil codes with overlapped tiling. *Concurrency and Computation: Practice and Experience*, 21(1):25–39, 2009.

[21] L. Haralick and L. Shapiro. *Computer and Robot Vision*. Addison Wesley, 1992.

[22] C. A. R. Hoare. Communicating sequential processes. *Communications of ACM*, 21(8):666–677, 1978.

[23] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP '05: Proceedings of the 2005 Workshop on Memory System Performance*, pages 36–43, New York, NY, USA, 2005. ACM.

[24] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[25] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[26] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. Technical report, Rice University, 1993.

[27] Ulrich Kremer. Np-completeness of dynamic remapping. In *In Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, pages 135–141, 1993.

[28] M. Krishna, A. Kumar, N. Jayam, and G. Senthilkumar. A synchronous mode MPI implementation on the Cell BE^{TM} architecture. *Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications*, 2007.

[29] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–244, New York, NY, USA, 2007. ACM.

[30] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, Jan 2002.

[31] D. J. Kuck, Y. Muraoka, and S.-C. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computing*, 21(12):1293–1310, 1972.

[32] D. L. Kuck. *Structure of Computers and Computations*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[33] A. Kumar, G. Senthilkumar, M. Krishna, N. Jayam, P. K. Baruah, R. Sharma, A. Srinivasan, and S. Kapoor. A buffered-mode MPI implementation for the Cell BETM processor. *ICCS '07: Proceedings of the 7th International Conference on Computational Science, Part I*, pages 603–610, 2007.

[34] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, New York, NY, USA, 1991. ACM.

[35] L. Lamport. The parallel execution of DO loops. *Communications of ACM*, 17(2):83–93, 1974.

[36] L. Lamport. The coordinate method for the parallel execution of iterative DO loops. Technical report, SRI, 1976.

[37] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[38] Peizong Lee and Zvi Meir Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Trans. Program. Lang. Syst.*, 24(1):1–50, 2002.

[39] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *J. Parallel Distrib. Comput.*, 13(2):213–221, 1991.

[40] D. B. Loveman. High performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, 1993.

[41] Mary E. Mace. *Memory storage patterns in parallel processing.* Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[42] K. Morton and D. Mayer. *Numerical Solution for Partial Differential Equations.* Cambridge Univ. Press, 2005.

[43] Y. Muraoka. *Parallelism exposure and exploitation in programs.* PhD thesis, Champaign, IL, USA, 1971.

[44] B. Palmer and J. Nieplocha. Efficient algorithms for ghost cell updates on two classes of MPP architectures. *14th IASTED International Conference on Parallel and Distributed Computing and Networks*, Jan 2002.

[45] C. J. Patten, H. A. James, K. A. Hawick, and A. L. Brown. Stencil methods on distributed high performance computers. Technical report, 1997.

[46] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[47] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, August 2007.

[48] L. Renganarayanan, M. Harthikote-Matha, R. Dewri, and S. V. Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *IPDPS*, pages 1–10, 2007.

[49] A. Sawdey, M. O'Keefe, R. Bleck, and R. W. Numrich. The design, implementation, and performance of a parallel ocean circulation model. In *Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology*, pages 523–550, 1994.

[50] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[51] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House Publishers, third edition edition, 2005.

[52] Marco Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.

[53] M. Velamati, A. Kumar, N. Jayam, and G. Senthilkumar. Optimization of collective communication in intra-cell MPI. *Lecture Notes in Computer Science*, pages 488–499, Jan 2007.

[54] B. Wilkinson and M. Allen. *Parallel Programming*. Pearson Prentice Hall, 2005.

[55] M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM.

[56] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.

[57] M. J. Wolfe. Techniques for improving the inherent parallelism in programs. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1978.

Index